Escuela Politécnica Superior de Jaén
Grado en Ingeniería Informática

Universidad de Jaén
Escuela Politécnica Superior
de Jaén

TRABAJO FIN DE GRADO

SEGMENTACIÓN INTELIGENTE DE OBJETOS EN NUBES DE PUNTOS

Alumno

Miguel Díaz Medina

Tutores

Carlos Javier Ogáyar Anguita
(Departamento de Informática)

José Manuel Fuertes García
(Departamento de Informática)

Junio, 2019

(Página intencionalmente en blanco)

Universidad de Jaén

Departamento de Informática

Don Carlos Javier Ogáyar Anguita y Don José Manuel Fuertes García, tutores del Trabajo Fin de Grado titulado: '**Segmentación inteligente de objetos en nubes de puntos**', que presenta Don Miguel Díaz Medina, otorgan el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, junio de 2019

El alumno: Los tutores:

Miguel Díaz Medina

Carlos Javier Ogáyar Anguita

José Manuel Fuertes García

(Página intencionalmente en blanco)

Agradecimientos

A mis padres, Miguel y Rocío, por creer en mí, por haberme apoyado en cada decisión que he tomado y por concederme el privilegio de estudiar esta carrera tan bonita. A mis tutores, José Manuel y Carlos, por darme la oportunidad de trabajar con ellos, por haberme enseñado tanto y por la ayuda prestada cada vez que la he necesitado. A mi familia, amigos y amigas, por haber tenido paciencia conmigo incluso en los días más difíciles. Gracias.

FICHA DEL TRABAJO FIN DE TÍTULO	
Titulación	Grado en Ingeniería Informática
Modalidad	Trabajo Teórico/Experimental
Especialidad (solo TFG)	Sistemas de Información
Mención (solo TFG)	Tratamiento Inteligente de la Información
Idioma	Español
Tipo	Específico
TFT en equipo	No
Autor/a	Miguel Díaz Medina
Fecha de asignación	09/04/2019
Descripción corta	Este trabajo consiste en la realización de un estudio teórico-práctico sobre la aplicación de técnicas basadas en aprendizaje profundo a la segmentación de información procedente de dispositivos de adquisición 3D (LiDAR, fotogrametría, Sfm, etc). La información adquirida genera conjuntos de datos a gran escala que son utilizados en ámbitos como la Ingeniería Civil, la Arquitectura (BIM), las ciudades inteligentes, la gestión del territorio (Topografía y cartografía) o la Arqueología. La **segmentación**, como paso previo a la clasificación e identificación de objetos, es una técnica muy utilizada tanto en 2D como en 3D. En 2D se han conseguido grandes avances, especialmente desde el campo de la Inteligencia Artificial. En cambio, para el caso 3D no han aparecido soluciones efectivas hasta hace unos pocos años, siendo el Aprendizaje Automático una de las técnicas más prometedoras, y en concreto, el **Aprendizaje Profundo**. El objetivo principal de este trabajo es realizar un estudio teórico de las principales técnicas basadas en **Aprendizaje Profundo** que se pueden aplicar a la segmentación semántica en nubes de puntos 3D, analizando varias alternativas del estado del arte de las principales técnicas, y experimentando sobre alguna en particular ofrecida por la literatura científica. Esto incluye el desarrollo de una arquitectura de red y la aplicación de la misma a problemas concretos dentro de los ámbitos descritos anteriormente, como por ejemplo datos procedentes de LiDAR o aplicaciones de Ingeniería Civil.

NORMAS APLICADAS EN ESTE DOCUMENTO	
LOCALES	
TFT-UJA:2017	Normativa de Trabajos Fin de Grado, Fin de Máster y otros Trabajos Fin de Título de la Universidad de Jaén (Normativa marco UJA aprobada en Consejo de Gobierno)
TFT-EPSJ:2017	Normativa sobre Trabajos Fin de Grado y Fin de Máster en la Escuela Politécnica Superior de Jaén (Normativa EPSJ aprobada en Junta de Escuela)
TFT-EPSJ	Criterios de evaluación y normas de estilo para TFG y TFM de la Escuela Politécnica Superior de Jaén
NACIONALES E INTERNACIONALES	
ISO 2145:1978	Documentación - Numeración de divisiones y subdivisiones en documentos escritos
UNE 50132:1994	Traducción de la ISO 2145
APA 6ª edición	Estilo de referencias y citas de APA (American Psychological Association)

NORMAS UTILIZADAS COMO BASE O REFERENCIA	
NACIONALES	
UNE 157001:2014	Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico
UNE 157801:2007	Criterios generales para la elaboración de proyectos de sistemas de información
*Estas normas se han utilizado **como base o referencia** para la inclusión de algunos contenidos y definiciones sobre elaboración de proyectos, entendiendo como **proyecto** la documentación consensuada entre una empresa y un cliente, que da lugar al perfeccionamiento de un contrato para la elaboración de una obra o la prestación de un servicio. Por consiguiente, no debe esperarse la aplicación de estas normas en cuanto a la completitud de los contenidos ni a la organización de los mismos.*	

Contenido

Índice de figuras

Índice de tablas

1 ESPECIFICACIÓN DEL TRABAJO

En este capítulo se presenta la especificación del trabajo, con una estructura y contenidos **inspirados** en los criterios y recomendaciones que establece la norma UNE 157801:2007 - "*Criterios Generales para la elaboración de proyectos de Sistemas de Información*".

A lo largo del documento se utilizarán términos y acrónimos cuya descripción aparecen en el apartado 6 (Definiciones y abreviaturas).

1.1 Introducción

Desde el momento en que el hardware de ordenadores evolucionó de tal manera que consiguió especializarse dando lugar a dispositivos de captura que nos permiten representar la componente espacial del mundo en un ordenador, surge la necesidad de hacer un tratamiento especial de esta información y, de ser capaces de obtener de manera automática patrones o características que se verifican en la realidad; que permitirían en consecuencia elaborar mejores soluciones para la misma, y en definitiva, realizar un tratamiento inteligente de la información espacial.

La llegada del coche autónomo (Bin Sulaiman, 2018), las *Smart Cities* (Ciudades Inteligentes) (Curry, Dustdar, Sheng, & Sheth, 2016), la gestión inteligente del territorio, el BIM (*Building Infomation Modeling*) en arquitectura, entre otras aplicaciones, necesitan de algoritmos capaces de realizar tratamiento automático de la información espacial para poder cumplir con su labor.

A modo de esclarecer esta breve introducción, el tipo de datos del mundo que van a estudiarse en este informe son datos espaciales (Bronstein, Bruna, LeCun, Szlam, & Vandergheynst, 2017). Cuando se trata de representar datos espaciales, es posible escoger entre una representación bidimensional, usando únicamente dos coordenadas (x, y) para representar la información (o longitud/latitud, etcétera); u optar por añadir una tercera dimensión, teniendo así, datos 3D (tridimensional), con tres coordenadas (x, y, z).

La tarea de **segmentación**, en lo que a datos espaciales se refiere, consiste en encontrar regiones de un plano o de un espacio que comparten características intrínsecas como el color, textura, reflectividad de la superficie de los objetos, etcétera. En el caso de una imagen (plano), en la que aparecen varios objetos, segmentar los objetos de esa imagen consistiría en encontrar qué **píxeles** de la imagen tienen características similares y distinguibles del resto. Si se trata de **segmentación semántica** (Shelhamer, Long, & Darrell, 2017), el proceso es similar con una **concepción semántica añadida**. Esto es, se conoce qué característica semántica comparten los píxeles: se trata de píxeles de una **mesa**, de una **televisión**, o los diferentes objetos que se hayan considerado **semánticamente**. Identificar el tipo de objeto forma parte de una tarea diferente: **clasificación** (Krizhevsky, Sutskever, & Hinton, 2012). Cuando se trata de datos tridimensionales, la tarea viene a ser lo mismo: **identificar regiones del espacio** con características similares. Como analogía a lo expuesto inmediatamente antes, la **imagen bidimensional** se convierte en una **escena tridimensional**. En cuanto a formatos de representación existentes, los datos tridimensionales no están tan limitados a una representación determinada como es el caso de datos bidimensionales (imagen o nube de puntos 2D), sino que coexisten varios tipos: nube de puntos, vóxel, malla, o incluso mediante imágenes. En este trabajo, el objeto de interés son las nubes de puntos como fuente de datos de partida.

Si se plantea la **segmentación** sobre una **nube de puntos tridimensional** (Huang & You, 2016; Maturana & Scherer, 2015; Charles R Qi, Su, Mo, & Guibas, 2016; Charles Ruizhongtai Qi, Yi, Su, & Guibas, 2017), aparecen varios retos debido a la alta redundancia y falta de estructura natural de las nubes de puntos (Charles R Qi et al., 2016). Una nube de puntos se construye sobre un espacio métrico en el que no existe un orden entre los puntos. Esto es, no puede establecerse un **operador** que ordene los puntos de alguna forma utilizando únicamente las propiedades del espacio métrico, como la **distancia** entre los puntos. El objetivo principal de la segmentación es simplificar y/o cambiar la representación de un modelo concreto por otro más significativo y más sencillo de analizar, dejando paso a otro tipo de algoritmos especializados en clasificación o detección de objetos, por ejemplo, para determinar qué objetos aparecen en la escena y/o qué posición ocupan.

El **objetivo principal** de este informe es realizar un **estudio teórico sobre las principales técnicas de segmentación en nubes de puntos 3D**, desde las técnicas clásicas hasta las más últimas y novedosas que utilizan inteligencia artificial para llevar a cabo la tarea. Además, se abordará el **desarrollo de una técnica de segmentación en nubes de puntos** con el objetivo de realizar una experimentación y obtener conclusiones.

1.2 Objetivos del trabajo

Con la realización de este proyecto **se pretende realizar un estudio teórico-práctico de las principales técnicas de segmentación en nubes de puntos que utilizan aprendizaje profundo**. Para ello, **se describirán teóricamente** las principales técnicas que están en el estado del arte, y **se seleccionará una** de ellas sobre la que se aplicará una adecuada experimentación.

Para establecer un contexto, se realizará un **repaso de los distintos tipos de representación** que pueden darse cuando se trabaja con datos 3D, pues no son las nubes de puntos la única solución, ya que existen algunas más, como ya se mencionó antes: vóxel, imagen, malla, etcétera.

Se aplicará una técnica basada en **Aprendizaje Profundo** para resolver la tarea. Este tipo de algoritmos componen un **subconjunto del Aprendizaje Automático**; en el contexto de la **Inteligencia Artificial**. Serán introducidos de manera concisa todos los conceptos necesarios para poder seguir este trabajo, dejando claro todo lo importante para tener una base sólida cuando se citen conceptos propios del Aprendizaje Profundo.

Una vez desarrollado todo el conocimiento necesario para abordar el proyecto, y expuestas algunas de las principales soluciones en el estado del arte, se experimentará con una técnica basada en la propuesta de (Huang & You, 2016). Se someterá esta técnica a diversas fases de entrenamiento y validación, realizando finalmente un conjunto de experimentos sobre la misma. A modo de cierre, se detallará un apartado de conclusiones, con nuevas propuestas abiertas a trabajos

futuros, y los aspectos más interesantes que se han observado con la experimentación.

Principalmente, se hará un estudio intensivo de una **técnica de segmentación que utiliza la voxelización como forma de tratamiento de la nube de puntos, con el objeto de obtener una representación regular que pueda servir de entrada a una red neuronal convolucional.**

1.3 Antecedentes y estado del arte

1.3.1 Aprendizaje Automático

El **aprendizaje automático** o ***machine learning*** es una disciplina dentro de la Inteligencia Artificial que consiste en crear programas capaces de aprender conforme a su experiencia (Russell & Norvig, 2004). A más aprendizaje (o experiencia), mejor es el rendimiento. Normalmente, la mejora del rendimiento se consigue en base a la aplicación del algoritmo sobre una serie de ejemplos de entrada que, en función de su naturaleza, determinarían el tipo de aprendizaje. En la primera iteración, la experiencia es nula y por tanto el rendimiento, por lo general, será bajo. A medida que se incremente el número de iteraciones, el rendimiento mejorará. En cierto modo, se trata de sistemas que se realimentan a sí mismos en base a una medida de rendimiento.

Se dan **3 tipos de aprendizaje** según el tipo de realimentación:

- **Aprendizaje supervisado.** Se trata de aprender una función a partir de ejemplos de sus entradas y sus salidas.
- **Aprendizaje no supervisado.** Consiste en aprender a partir de ejemplos de entrada para los que no se especifica su valor de salida.
- **Aprendizaje por refuerzo.** El agente de aprendizaje por refuerzo debe aprender cómo se comporta el entorno mediante recompensas (refuerzos) o castigos, derivándose del éxito o fracaso respectivamente.

El aprendizaje automático pretende resolver unos tipos de problemática específicos: clasificación, regresión, regresión ordinal y agrupamiento (o *clustering*).

De entre todos los algoritmos de aprendizaje automático, interesan los **algoritmos basados en redes neuronales:** Aunque el modelo matemático de neurona (McCulloch & Pitts, 1943) se propuso en la década de 1940, no fue hasta años más tarde cuando apareció un método matemático que permitía entrenar estos modelos (Cun, 1988). Básicamente, estos algoritmos intentan simular el procesamiento neuronal del cerebro biológico, si bien su aprendizaje es muy costoso y requiere unas capacidades de cómputo muy altas tanto en hardware como en software.

Estos algoritmos han demostrado ser muy eficaces tanto en tareas de clasificación como regresión, aplicándose en múltiples campos como visión artificial, reconocimiento automático del habla, etcétera. El modelo más básico de red neuronal es el perceptrón (véase Figura 1.1), compuesto por una única neurona. Se requieren unos conocimientos matemáticos muy consolidados para poder trabajar con estas redes con destreza. Al tratarse de uno de los pilares de este proyecto, se dedicará una sección de este documento a su completa descripción y entendimiento, pasando por sus elementos básicos, el entrenamiento, y las principales arquitecturas.

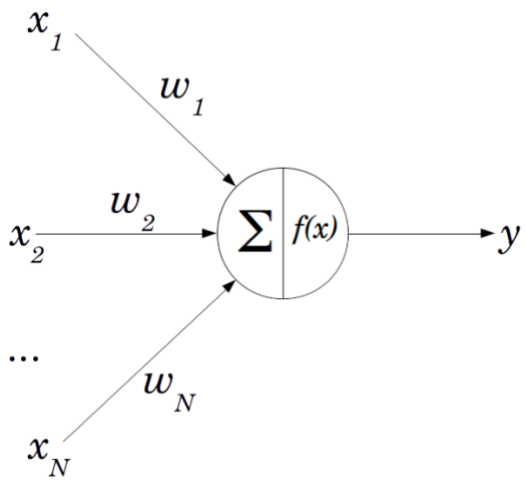

Figura 1.1: Perceptrón. Fuente: Lucidarme

Aunque existen diferentes medidas de rendimiento para **evaluar** la calidad de los modelos obtenidos (precisión, cobertura, curvas ROC, matrices de confusión, …) en diferentes algoritmos de aprendizaje automático, un aspecto particular de las

redes neuronales, que permite decidir si escoger o no este esquema sobre el resto, es la baja **interpretabilidad** del modelo entrenado. Dicho concepto se define como la capacidad que tiene un modelo de ser revisado por un **experto** y verificar la calidad del conocimiento extraído.

Como se ha mencionado en el párrafo anterior, las redes neuronales tienen una interpretabilidad muy baja o nula. Se dispone de un modelo formado por un conjunto de pesos y valores de sesgo (datos numéricos, lo cual no aporta nada si no es posible establecer una semántica entre ellos) y en cantidades que superan los cientos de millones. Es imposible (o difícilmente practicable) para un experto en el dominio de aplicación, revisar que el conocimiento extraído es de calidad; la única medida de calidad existente es el tradicional ciclo de vida de prueba-error durante las fases de entrenamiento/validación/test.

Otro aspecto a tener en cuenta al trabajar con algoritmos de aprendizaje es el **sobreajuste** o **sobreaprendizaje**. Los datos tienen dos partes, **1) señal**, la distribución de probabilidad que siguen los ejemplos del conjunto de datos, y **2) ruido**, producto de las imprecisiones de los dispositivos de adquisición o bien la presencia de *outliers* en el conjunto de datos; un objetivo ideal para el algoritmo de aprendizaje sería capturar únicamente la **señal**, esto es, la distribución de probabilidad que siguen los datos de partida y así hacer predicciones correctas sobre datos de test que también sigan la misma distribución. En la mayoría de situaciones, hay ejemplos en el conjunto de datos que no siguen la misma distribución al resto, es a esto a lo que se conoce como **ruido**. Estos ejemplos empeoran el entrenamiento, pues si el modelo consigue ajustarse de tal forma que haga predicciones correctas sobre estos datos anómalos, entonces habrá **sobreaprendido** las peculiaridades del conjunto de entrenamiento, y no solamente ha capturado la señal de los datos de entrada sino que también ha aprendido a trabajar con datos que no siguen esa misma distribución de probabilidad, resultando en un modelo entrenado erróneamente al momento de hacer predicciones sobre datos que no se han contemplado durante el entrenamiento. Esta caracterización de los datos es difícilmente tratable, pues es difícil distinguir a simple vista cuál es la señal y cuál es el ruido. La manera de hacer esto es mediante ciclos de entrenamiento y validación con particiones disjuntas sobre el conjunto original. El

sobreaprendizaje se trata, pues, de una medida del ajuste del modelo a los datos, esto es, en qué medida el modelo ha conseguido aprender sobre los datos de entrenamiento. Si un modelo sobreajusta los datos de entrenamiento, entonces no ejercerá una tarea predictiva correcta sobre el conjunto de test.

1.3.2 Aprendizaje Profundo

El **Aprendizaje Profundo** o ***Deep Learning*** es un nuevo paradigma de aprendizaje automático (véase la relación en Figura 1.2) que científicamente se ha hecho muy popular en los últimos años y sobre el que no cesan de aparecer nuevas aportaciones en la literatura especializada. A diferencia del aprendizaje automático tradicional, en el que se debe construir una especificación formal de todo el conocimiento que un ordenador necesita, este nuevo esquema parte de comenzar la tarea a partir de conceptos simples, que vayan relacionándose entre sí, y con esto construir una jerarquía de conceptos más complejos a partir de los más simples (Goodfellow, Bengio, & Courville, 2016). Si se construyese un grafo a partir de cómo los conceptos más complejos se posicionan por encima de los más simples, este grafo sería muy **profundo**. Es por esto por lo que se decidió llamar a este enfoque Aprendizaje Profundo o *Deep Learning*.

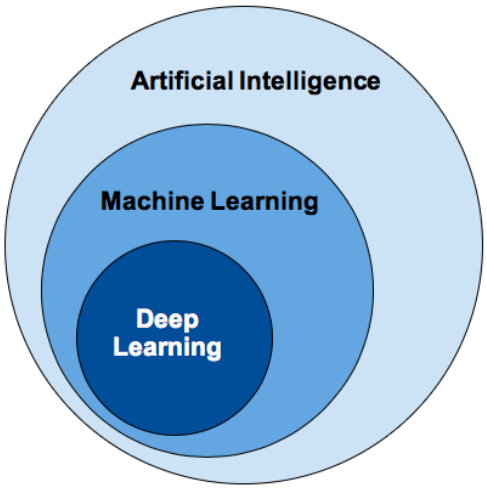

Figura 1.2: Ámbito del Deep Learning. Fuente: Sumologic

Se plantea una necesidad de que los ordenadores extraigan su propio conocimiento, con su propia representación del mismo, y utilicen su propio esquema de razonamiento, evitando utilizar de manera estricta el conocimiento que el humano

experto ha introducido formalmente en el ordenador que, probablemente, no se encuentre en una forma óptima para realizar la tarea (véase capítulo introductorio de Goodfellow et al., 2016: *Introduction. Representation learning*). A esta nueva forma de obtener conocimiento es a lo que se conoce como Aprendizaje Profundo o *Deep Learning*. El rendimiento de los algoritmos de *machine learning* depende fuertemente de la representación utilizada para los datos. La gran dificultad en las aplicaciones de inteligencia artificial es que muchos factores de variación influyen sobre cada pieza de los datos que somos capaces de observar. Esto se refiere a que, por ejemplo, en clasificación de coches, es prácticamente imposible obtener datos de coches de la misma perspectiva, el mismo color, mismo tamaño, etcétera, por lo que necesitamos características de más alto nivel (una representación de alto nivel) para poder realizar la tarea de aprendizaje con éxito, como por ejemplo la presencia de objetos determinados en imágenes. Para poder aprender estas características, se necesita un entendimiento humano muy sofisticado de alto nivel.

Es aquí donde el aprendizaje profundo cobra sentido y permite resolver este problema central de la representación introduciendo representaciones que son expresadas en términos de conceptos más simples. Por ejemplo, en la Figura 1.3 se observa cómo un sistema de aprendizaje profundo puede representar el concepto de la imagen de una persona combinando conceptos simples como contornos y esquinas:

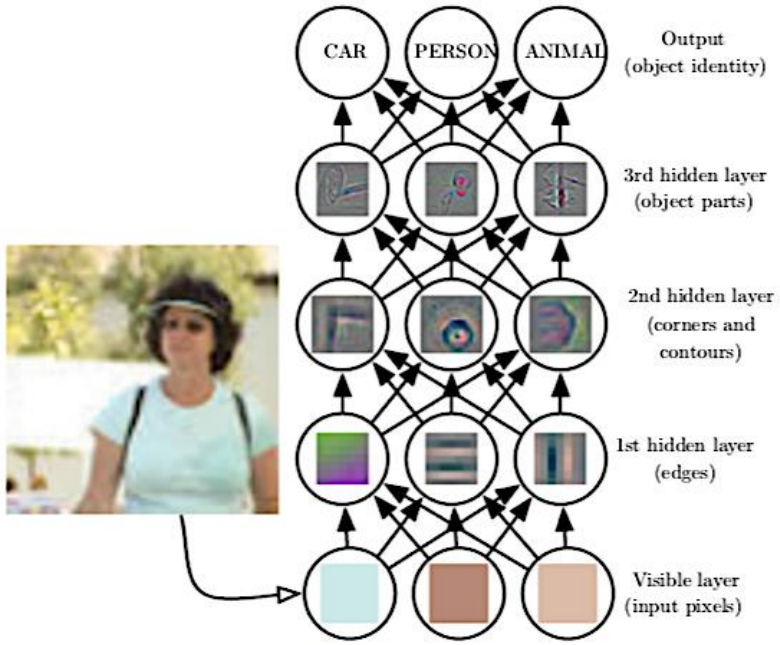

Figura 1.3: Representación de cómo un sistema de aprendizaje profundo aprende a construir conceptos complejos a partir de los más simples. Fuente: Goodfellow et al., 2016

El ejemplo por excelencia de un modelo de aprendizaje profundo es la red neuronal **Perceptrón Multicapa o MLP *(Multi-Layer Perceptron)*** (véase Figura 1.4)**.** Se trata de un aproximador para una función matemática que mapea algún(os) valor(es) de entrada a valor(es) de salida. La función resulta de la composición de otras funciones mucho más simples. Puede verse la aplicación de cada función matemática como un intercambiador de la representación de los datos de entrada.

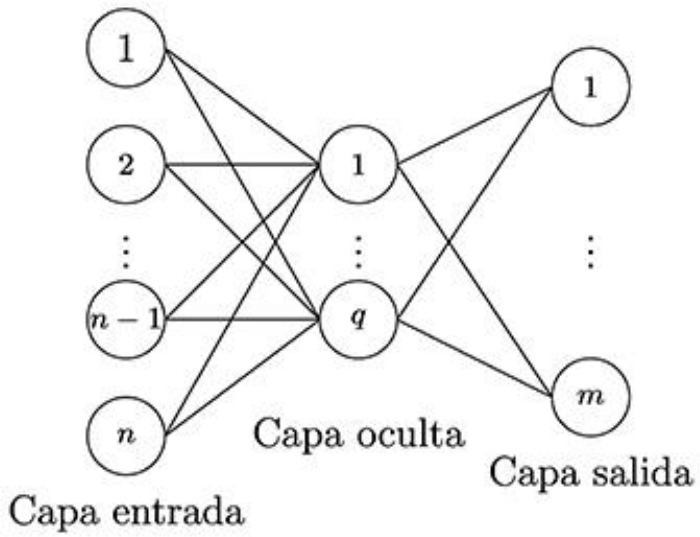

Figura 1.4: Perceptrón Multicapa. Fuente: Scielo

La idea de aprender la representación adecuada para los datos provee una perspectiva para el aprendizaje profundo. Otra perspectiva sobre el aprendizaje profundo es que la **profundidad** en capas permite que el sistema artificial aprenda un programa de varios pasos. Cada capa de la representación puede considerarse como el estado de la memoria en el ordenador después de ejecutar un conjunto de instrucciones en paralelo. Las redes más profundas pueden ejecutar más instrucciones de forma secuencial. Las instrucciones secuenciales ofrecen una gran potencia porque instrucciones posteriores pueden referirse a resultados generados en etapas anteriores (Goodfellow et al., 2016).

A modo de conclusión, y antes de comenzar con regresión logística como precursora del aprendizaje profundo, entiéndase este paradigma como una forma de obtener conocimiento de más alto nivel en una representación difícil de introducir formalmente en un ordenador, de una manera similar a cómo identifican **patrones** los humanos. Se trata de un tipo particular de aprendizaje automático que consigue una gran potencia y flexibilidad en la manera en que aprenden a representar el mundo como una jerarquía de conceptos relacionados, con cada concepto definido en relación a otros conceptos más simples, y representaciones más abstractas procesadas en torno a las menos abstractas.

1.3.2.1 Regresión logística

Se trata de una de las técnicas de aprendizaje más empleadas hoy en día. En multitud de ocasiones, la aplicación de un modelo de regresión lineal en problemas de clasificación no es adecuada puesto que el espacio de muestras no es **separable linealmente** (véase Figura 1.10**)**.

Pese a su nombre (regresión), se trata de una técnica de clasificación, donde la función aprendida no es lineal. Intuitivamente, se trata de aprender una función que permita clasificar entre dos tipos de clases; la salida es una probabilidad estimada $[0,1]$, y podría asignarse a la **clase A** todo lo que esté por debajo de 0.5, y a la **clase B** todo lo que esté por encima.

Ha de representarse una hipótesis estadística, la cual debe aprenderse.

$$h_\theta(x) = g(\theta^T x) \tag{1.1}$$

Donde *x* es el dato a clasificar, θ es la matriz de parámetros de la regresión logística y *g(z)* es la función **Sigmoide** $g(z) = \frac{1}{1+e^{-z}}$ (véase Figura 1.5), una función no lineal capaz de **mapear** cualquier dato de entrada al intervalo $[0,1]$.

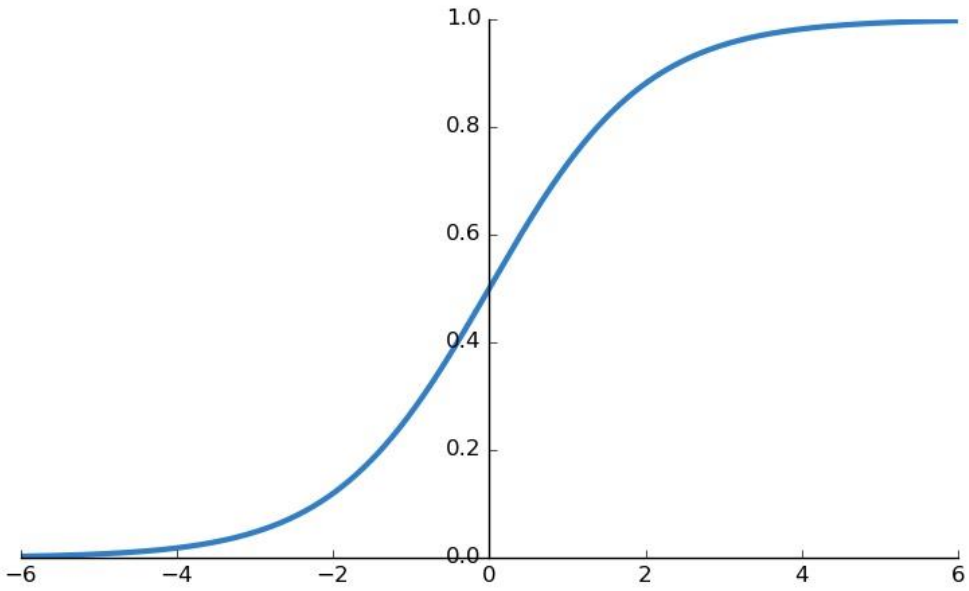

Figura 1.5: Función Sigmoide. Fuente: ResearchGate.

Como se aprecia en Figura 1.5, para cualquier valor de entrada, la función Sigmoide mapea al intervalo $[0,1]$. Con la regresión logística, se pretende resolver el enfoque desde una perspectiva probabilística:

$$h_\theta(x) = P(y = 1 \mid x;\ \theta) \tag{1.2}$$

Donde $h_\theta(x)$ representa la probabilidad de que el dato x sea clasificado como *1* con una hipótesis de parámetros θ. Para obtener la probabilidad de que sea clasificado como 0, simplemente hay que restar esta probabilidad a 1: $P(y = 0 \mid x;\ \theta) = 1 - P(y = 1 \mid x;\ \theta)$.

El **límite de decisión** constituye una frontera (línea) definida en el espacio de muestras que separa los ejemplos según su clase, comportándose como un **hiperplano** de separación. El objetivo de la regresión logística es obtener el límite de decisión que mejor separa las clases. Esta línea no tiene por qué ser una recta; de hecho, en la mayoría de los casos no lo es, sino que es una curva, una circunferencia, etcétera.

Cada uno de los atributos del dato x de entrada tiene que verse modificado por los parámetros en θ: un vector de parámetros. Es impracticable encontrar una configuración para estos parámetros de forma manual, pues el tamaño de este vector podría hacerse demasiado grande, por lo que esto ha de resolverse automáticamente.

Se parte de un conjunto de datos de entrada donde cada ejemplo (*ítem*) de ese conjunto está etiquetado con su clase. Ha de definirse una función de coste que mida el error cometido por el clasificador al operar sobre cada ejemplo, por comparación con el valor que realmente tiene asociado ese ejemplo de entrada. Se define un problema clásico de optimización: encontrar el mejor valor (óptimo global) para los parámetros que permiten clasificar los datos de forma correcta.

Puede definirse una función de coste como:

$$J(\theta) = \frac{1}{m}\sum_{i=0}^{m} Coste(h_\theta(x)^{(i)}, \qquad y^{(i)}) \tag{1.3}$$

donde m es el número total de ejemplos en el conjunto de entrenamiento, $y^{(i)}$ es el valor real de etiqueta asociada al ejemplo i, y $h_\theta(x)^{(i)}$ es el valor estimado por la regresión logística para el ejemplo i.

Mediante la **derivación** (cálculo de la derivada parcial con respecto a cada parámetro $\theta_j \in \theta$) de esta función de coste (no se va a describir la funcionalidad del operador $Coste(h_\theta(x)^{(i)}, y^{(i)})$ por simplicidad en la explicación), se puede estimar el error cometido por cada parámetro θ_j en θ, y actualizar cada uno de ellos conforme

a ello, para intentar minimizar el error. Esto es lo que se conoce como **Gradiente Descendente**.

$$\theta_j := \theta_j - \alpha * \frac{d}{d\theta_j} J(\theta) \tag{1.4}$$

donde θ_j representa a cada peso del vector de pesos θ con índice j; α indica el ratio de aprendizaje en cada iteración, esto es, la medida en la que cada peso se ve afectado según el error cometido; y $\frac{d}{d\theta_j} J(\theta)$ representa la derivada parcial de la función de coste con respecto a cada uno de los parámetros que componen el vector de pesos.

Mediante la aplicación del operador anterior se actualiza cada parámetro $\theta_j \in \theta$, intentando minimizar el error cometido por el estimador hasta encontrar la configuración de parámetros que minimiza la función de coste, el óptimo global. Ante cualquier problema de optimización existe el riesgo de caer en un óptimo local. La derivada de la función permite calcular qué dirección (el vector gradiente) se debe tomar para conseguir minimizar el error, mientras que el ratio de aprendizaje evita que los saltos en el espacio de soluciones sean demasiado bruscos y sea más probable encontrar el óptimo global.

La **regresión logística** no es más que la precursora de todo el conocimiento y funcionamiento bajo las redes neuronales, asociándose cada neurona como una **unidad de regresión logística**. Una **red neuronal** es, pues, una asociación masiva de unidades de regresión logística interconectadas entre ellas. En el apartado siguiente se describirán todos los aspectos relevantes de las redes neuronales.

1.3.2.2 Redes Neuronales

Una **red neuronal** es un modelo computacional basado en un gran conjunto de unidades neuronales simples (unidades de regresión logística) de forma aproximadamente análoga al comportamiento observado en los axones de las neuronas de un cerebro biológico (Schmidhuber, 2015).

Pueden contener una o varias capas ocultas; más aún en aprendizaje muy profundo. Cada **capa** está formada por varios nodos o neuronas, y cada uno de los nodos se conecta a cada nodo de la capa siguiente. Así, hasta obtener un grafo dirigido acíclico en el que todas las neuronas de una capa están conectadas con todas las neuronas de la capa siguiente. Se dispone de una información que se introduce en la red a través de la capa de entrada y es procesada en la red, atravesando todas las capas hasta llegar a la capa de salida.

Toda capa de la red que no sea de entrada o salida se agrupa dentro del conjunto de capas ocultas. En la Figura 1.6 se observa el ejemplo por antonomasia de una red neuronal, en el que se cuenta con una capa de entrada, una capa oculta (aunque normalmente habrá varias) y una capa de salida.

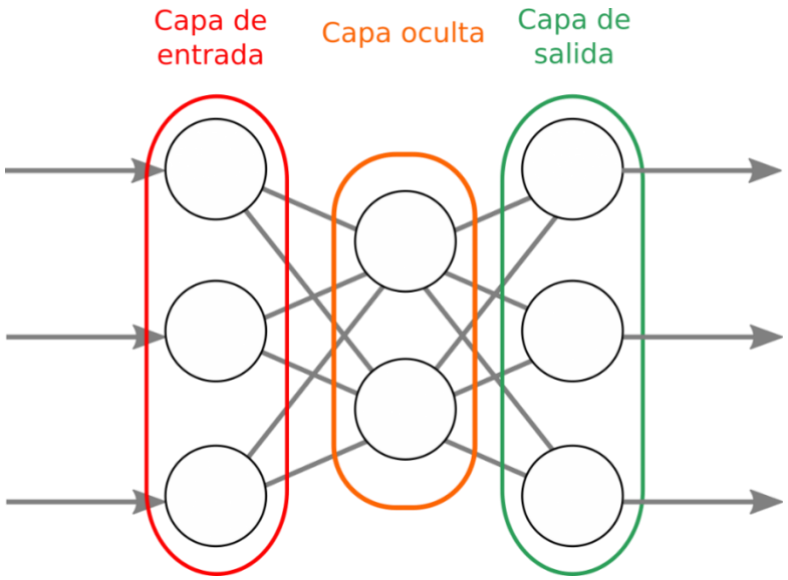

Figura 1.6: Prototipo de red neuronal. Fuente: Robologs

Cada unión entre dos neuronas representa un peso sináptico, $\boldsymbol{w_i}$. Dependiendo de cuál sea el objetivo de una red como modelo predictivo, la capa de salida de la misma será configurada de una manera determinada. Por ejemplo, en tareas de **clasificación**, habrá tantas neuronas en la capa de salida como clases entre las que distinguir tenga el problema. Por otra parte, en **regresión numérica**, en la capa de salida de la red habrá una única neurona, proporcionando un valor numérico v, donde $v \in \mathbb{R}$.

En definitiva, el objetivo de la red neuronal es aprender una función matemática: una hipótesis que puede ser lineal o no lineal, que mapea para cada valor de entrada X un valor de salida Y. $f: X \rightarrow Y$. X puede estar compuesto por un valor único, o ser un vector de valores (características) de entrada, al igual que Y. Con un entrenamiento apropiado y con un ajuste correcto de los **pesos sinápticos**, es posible obtener una red neuronal que mapea satisfactoriamente cada valor de entrada a un valor de salida según la función aprendida, independientemente de la tarea.

Para realizar el entrenamiento se necesitan datos etiquetados, esto es, datos con el valor real Y asociado al valor X de entrada (aprendizaje supervisado). A grandes rasgos, el proceso de entrenamiento consistirá en: introducir en la red un dato del conjunto de entrenamiento y propagar hasta obtener un valor en la salida; es entonces cuando se compara este valor predicho con la etiqueta real (*ground truth*) asociada al dato de entrada, y se obtiene una medida de error que servirá para calcular el error cometido en cada capa, propagando hacia atrás, y con esto actualizar los pesos para disminuir el error en la siguiente iteración.

Las **redes neuronales** son modelos computacionales extremadamente complejos, con un alcance y ámbitos de aplicación aún por conocer. Esta complejidad inherente conlleva a que trabajar con estos modelos sea computacionalmente muy costoso, y que sea necesario un esfuerzo considerable en ajustar correctamente todos los **hiperparámetros** para que el modelo sea entrenado correctamente.

Además, y no menos importante, se necesita de un hardware especializado muy potente como son las **GPU** (*Graphics Processing Unit*), ya que este tipo de procesadores adoptan una arquitectura SIMD (*Simple Instruction Multiple Data*) vectorial, ideal para realizar operaciones con matrices. En las arquitecturas SIMD, se puede realizar la misma operación sobre distintos conjuntos de datos, esto es, pueden ejecutarse varias hebras en paralelo. Una sola propagación de la red, es decir, tomar como entrada un ejemplo y generar un valor de salida, conlleva multiplicaciones de matrices de dimensionalidad alta. Esta operación es altamente ineficiente en la **CPU** (*Central Processing Unit*), ya que una sola hebra debe calcular

cada posición de la matriz resultante. En cambio, una GPU permite que existan tantas hebras como posiciones tenga la matriz de salida, y todas ellas se ejecuten en paralelo, realizando la operación de una forma muy eficiente. El cálculo del gradiente en cada capa es también optimizado con esta arquitectura, resultando en un punto a favor de la optimización de los parámetros de la red.

En las siguientes secciones se detallará todo lo que se necesita saber sobre redes neuronales para abordar este proyecto, con el objeto de entender las operaciones que están sucediéndose durante el entrenamiento/test de las mismas, o cuál es el modo de *'pensar'* que tiene una red neuronal para realizar las diferentes tareas de clasificación/regresión.

Han sido varios los esquemas de redes neuronales profundas propuestas:

- **Redes Neuronales Convolucionales.** Surgen a raíz de la clasificación de imágenes, para evitar de parámetros que se generaban al diseñar una red en la que hay una unidad de entrada por cada píxel en cada canal de la imagen (Szegedy et al., 2015). Con esta nueva arquitectura, los pesos se comparten para todos los píxeles de la imagen (2D) o vóxeles del objeto (3D). Primero se hace una extracción de características sobre los datos de entrada, a modo incremental y jerárquico, comenzando por características básicas como las esquinas, bordes, entre otras; obteniendo cada vez características de más alto nivel, hasta concluir con una red neuronal estándar tal y como se ha descrito. Se dedicará una sección completa a describir esta arquitectura (véase Figura 1.7), ya que es uno de los ejes de este proyecto.

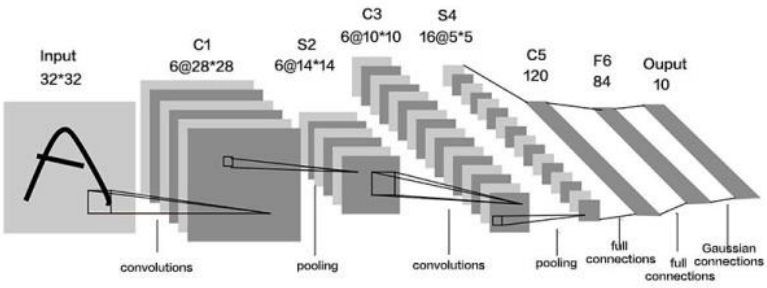

Figura 1.7: Prototipo de Red Neuronal Convolucional. Fuente: ResearchGate

- **Redes Generativas Adversarias.** El objetivo de este modelo es aprender la distribución de los datos del conjunto de entrenamiento, y generar nuevos datos con la misma distribución (Goodfellow et al., 2014). Por ejemplo, generar imágenes de caras de personas que no existen. La filosofía de trabajo de estas redes se encuadra como una competición en la que una red intenta engañar a otra continuamente. Existe una red discriminadora, y otra generadora. El trabajo de la red discriminadora es aprender a detectar si un dato de entrada es real (proviene del conjunto de entrenamiento) o falso (ha sido generado en la red generadora), mientras que el trabajo de la red generadora es alimentarse de la decisión del discriminador para aprender una función matemática que se ajusta dentro de un espacio multidimensional a la distribución que siguen los datos del conjunto de entrenamiento. Este segundo modelo recibe como entrada un elemento llamado **vector latente**: un conjunto de números generados aleatoriamente, que producirán una salida en el modelo generado. De alguna forma, la red discriminadora solo servirá para entrenar a la red generadora, y una vez esta se haya entrenado correctamente, dejará tener sentido. En la Figura 1.8 se observa una esquematización de esta novedosa arquitectura.

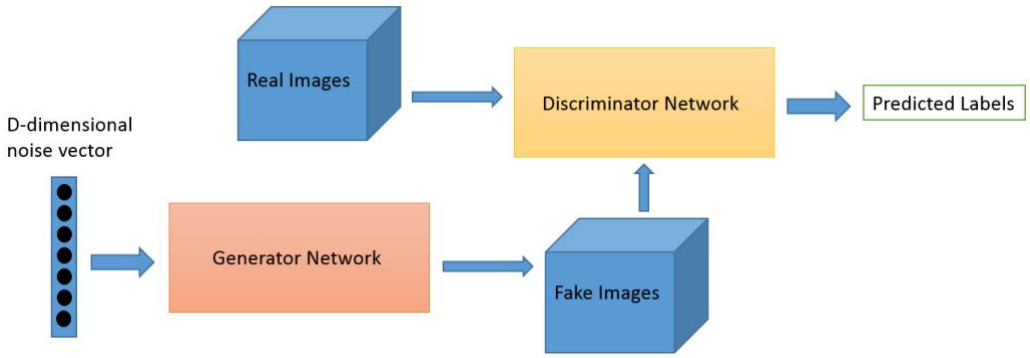

Figura 1.8: Prototipo de Red Generativa Adversaria. Fuente: Towards Data Science

- **Redes Neuronales Recurrentes.** Una red neuronal recurrente surge de la necesidad de estudiar modelos de datos en los que existe una relación de dependencia, es decir, modelos de secuencia (Jain & Medsker, 1999). El ejemplo más típico es el reconocimiento de voz, reconocimiento de entidades en un texto, etcétera. Se trata de modelos en los que la entrada

y la salida no está configurada de forma fija, esto es, no existe un número predeterminado de neuronas de salida, y puede ir variando (véase Figura 1.9). Si se recibe una secuencia de duración 10 segundos o 10 minutos, por ejemplo, la red podrá trabajar de la misma manera. Reciben como entrada tanto el dato a propagar, como el resultado de la propagación anterior. Estas redes llevan asociada una memoria inherente, dado que, para estudiar relaciones entre los datos de entrada es necesario recordar ciertos aspectos sobre los datos que se han introducido antes. Los modelos más típicos son GRU (*Gated Recurrent Unit*) o LSTM (*Long-Short Term Memory*). Pese a que existe una densa literatura y casos de uso de esta arquitectura, no va a profundizarse más en este campo puesto que no es objeto de estudio y entrar en su descripción no serviría de nada en el entendimiento de este proyecto.

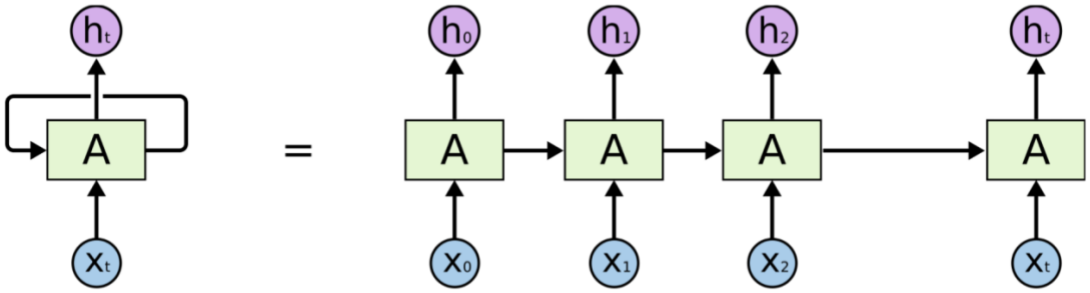

Figura 1.9: Prototipo de Red Neuronal Recurrente. Fuente: Medium

A continuación, se describe en detalle la manera en la que se construye una red neuronal.

Arquitectura funcional

Una red neuronal consta de una serie de elementos básicos a nivel de diseño como son: la capa de entrada, la(s) capa(s) oculta(s), y la capa de salida. Se describe cómo se configura cada una de ellas (Goodfellow et al., 2016).

- **Capa de entrada.** En la capa de entrada, habrá tantas neuronas como variables tenga el problema. Teniendo en cuenta que estas variables podrán ser de diferentes tipos (booleano, categórico, numérico), no siempre se cumplirá la correspondencia N variables → N neuronas de entrada. Si se trata de una variable booleana o numérica, servirá con una sola neurona de entrada para esa variable, pudiendo tomar los valores 0/1 según falso/verdadero si es booleana. En el caso de las variables categóricas, habrá tantas unidades de entrada como posibles valores pueda tomar esa variable, y de ese subconjunto solo una de ellas tomará el valor 1, mientras que el resto recibirán un cero.
- **Capas ocultas.** La configuración de las capas ocultas es algo que continúa siendo objeto de estudio por investigadores de todo el mundo, puesto que aún no está claro cómo tiene que hacerse. El número de capas tiene que ver con la abstracción que realiza la red sobre las características de entrada, generando sucesivamente conocimiento más abstracto y de más alto nivel con el que se pueda obtener la representación más adecuada de la función objetivo. El número de neuronas de cada capa tiene que ver con la dimensionalidad del espacio en el que se está buscando una frontera de separación. Se tratará esto a continuación. En definitiva, para configurar adecuadamente tanto el número de capas ocultas como el número de neuronas en cada una de ellas se basa en la complejidad de los datos de entrada, en su representación, generando un conjunto de parámetros sobre los que luego se ejecutarán heurísticas de búsqueda (búsqueda local, algoritmos evolutivos, algoritmos meméticos, etcétera), para encontrar la configuración de red óptima a nuestro problema.
- **Capa de salida.** En la capa de salida sucede algo similar a lo que sucede en la capa de entrada. Según el problema, se configurará de una manera u otra. En **clasificación**, por ejemplo, habrá tantas neuronas en la salida como clases distinguibles en el problema, y la neurona que se active será la clase predicha para el ejemplo de entrada. En tareas de **regresión**, solo habrá una neurona de salida.

Una vez descrita la configuración de los elementos más básicos que constituyen una red neuronal, las neuronas, ha de explicarse cuál es el funcionamiento de una red neuronal.

Una **red neuronal** recibe un dato como entrada, y genera una salida para ese dato. El formato en el que se representa tanto la entrada como la salida es numérico, si bien no es un único número, sino un conjunto de ellos. Se utiliza una representación matricial para representar un dato, de esta forma, en una sola propagación se alimentan todas las entradas de la red con todas las características o valores de entrada. En el momento de generar la salida, vuelve a obtenerse un vector de números, con tantos de ellos como salidas tenga la red.

La representación **matricial** es idónea cuando de redes neuronales se trata, ya que las matrices permiten trabajar con grandes cantidades de datos de manera simultánea. Los **pesos sinápticos** y **sesgos** se representan usando matrices, para facilitar la multiplicación/suma con los datos de entrada. Aparece un término que no había aparecido hasta ahora, el **sesgo**. Suele representarse con la letra b minúscula, y se utiliza para que la red sea capaz de aprender cualquier tipo de función matemática. Por ejemplo, la función NAND, cuya tabla de verdad es la siguiente (véase Tabla 1.1):

A	B	F(A, B)
0	0	1
0	1	1
1	0	1
1	1	0

Tabla 1.1: Función NAND

La pregunta es la siguiente, ¿cómo puede encontrarse una configuración de la red que proporcione un 1 cuando las dos entradas son 0? Con el paradigma actual, no es posible. Por tanto, **es necesario introducir un término de sesgo que proporcione un valor de salida aun cuando todas las entradas sean 0**.

Hecho este paréntesis, se continúa con la representación matricial de los datos.

$$X = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad Y = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \tag{1.5}$$

X, Y, representan respectivamente a un dato de entrada y a un dato de salida, mientras que n, m son respectivamente, el número de neuronas en la capa de entrada/salida. Por tanto, aunque se representará una función de una unidad neuronal de la siguiente manera:

$$Y = W^T * X + b \tag{1.6}$$

Es necesario saber que cada elemento de esa ecuación representa a una matriz. De hecho, W se representa como su traspuesta. Siendo W la matriz de pesos, en cada fila se tienen los pesos de todas las uniones de las neuronas de la capa anterior, con la neurona de la fila correspondiente. Si n es el número de neuronas de la capa L^i, y m es el número de neuronas de la capa L^{i+1}, la dimensión de la matriz W será $W_{n\,x\,m}$, y la dimensión de la matriz b será $b_{m\,x\,1}$.

Una neurona es capaz de realizar divisiones lineales del espacio de características definido por los datos de entrada, y encontrar fronteras de separación de naturaleza lineal, así como aproximar datos que sigan una distribución lineal en el espacio.

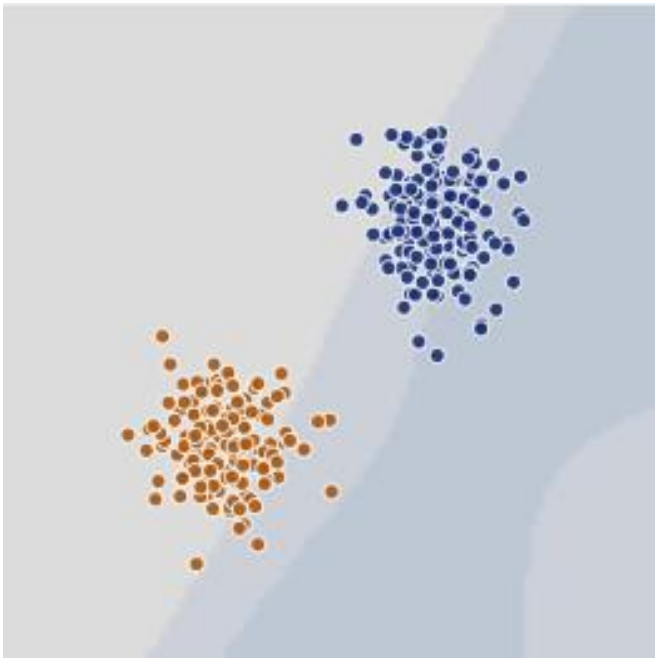

Figura 1.10: Datos separables linealmente. Fuente: Playground Tensorflow

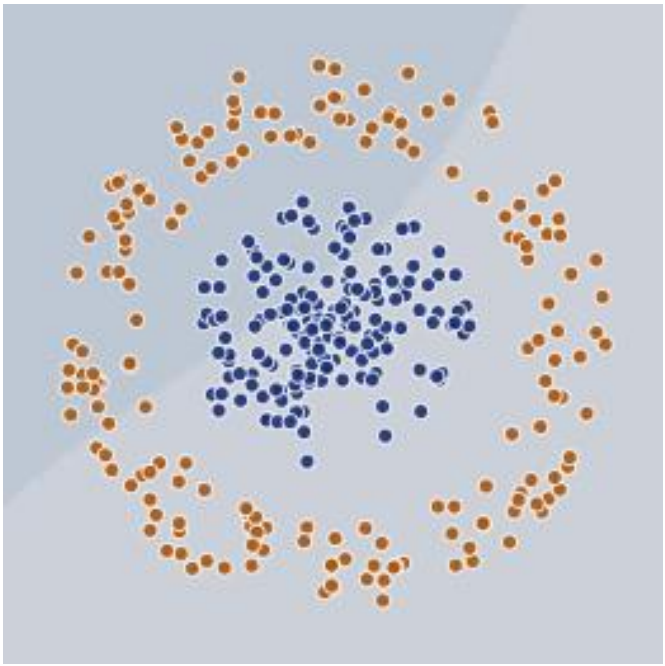

Figura 1.11: Datos no separables linealmente. Fuente: Playground Tensorflow

Si se observan las imágenes, se tienen datos clasificables en 2 clases diferentes. En la Figura 1.10 es posible dividir las clases utilizando una línea recta. En cambio, en el segundo caso (Figura 1.11) no es trivial, pues sería necesario trazar una **curva cerrada simple** (circunferencia). Es aquí donde cobran sentido las redes neuronales con sus respectivas funciones de activación a nivel de neurona

(capa), permitiendo encontrar límites de decisión de naturaleza no lineal, aproximando prácticamente cualquier función matemática.

A grandes rasgos, la asociación de múltiples neuronas conlleva múltiples deformaciones del espacio, obteniendo espacios 3, 4, 5, …, n-dimensionales, donde al final, tras realizar la propagación y encontrarnos ante la última capa, cada neurona de esta capa aprenda a dividir este espacio multidimensional en dos mitades, donde una mitad encerrará a todos los ejemplos que sean clasificables en la clase que se corresponda con esa neurona, mientras que la otra mitad representará a ejemplos que no son clasificables en esa clase.

Funciones de activación

La aproximación de funciones no lineales no es algo trivial, y hacen falta unos operadores conocidos como **funciones de activación**. Una vez se han multiplicado los datos de entrada por los pesos de la capa L y sumado el sesgo de esa capa, se aplica una nueva función (normalmente no lineal) que toma como entrada el valor obtenido, esto es, **composición** de funciones. El motivo, como ya se ha indicado, es darle la posibilidad a la red para que aprenda cualquier tipo de función (lineal o no lineal). La salida de estas funciones es un valor de activación, de la misma manera que ocurre en un cerebro biológico. Así, se puede controlar el hecho de que, si en una neurona se produce un valor negativo, este valor no se propague hacia adelante impidiendo o desfavoreciendo el ajuste de los pesos, proporcionándose un valor 0. Funciones de activación disponibles:

- **Sigmoide.** Esta función es natural de la regresión logística (véase Figura 1.12). Mapea el espacio de entrada a un rango $[0,1]$ de valores positivos. Suele utilizarse muy comúnmente, sobretodo en la última capa en tareas de clasificación, interpretando el valor como una probabilidad $[0,1]$ de que el ejemplo pertenezca a la clase predicha.

$$g(z) = \frac{1}{1 + e^{-z}} \tag{1.7}$$

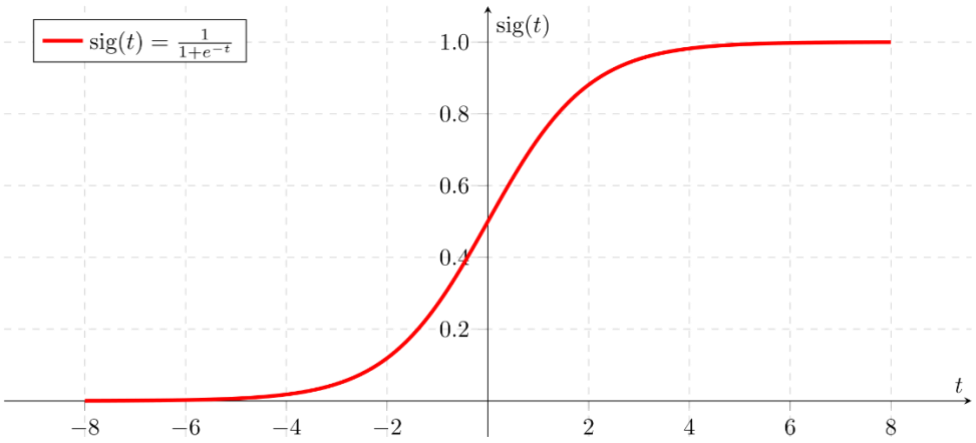

Figura 1.12: Sigmoide. Fuente: ResearchGate.

- **Tangente hiperbólica.** Sigue la misma filosofía que la función Sigmoide, con la diferencia de que el mapeo es al rango $[-1, 1]$ (véase Figura 1.13).

$$g(z) = \tanh(z) \tag{1.8}$$

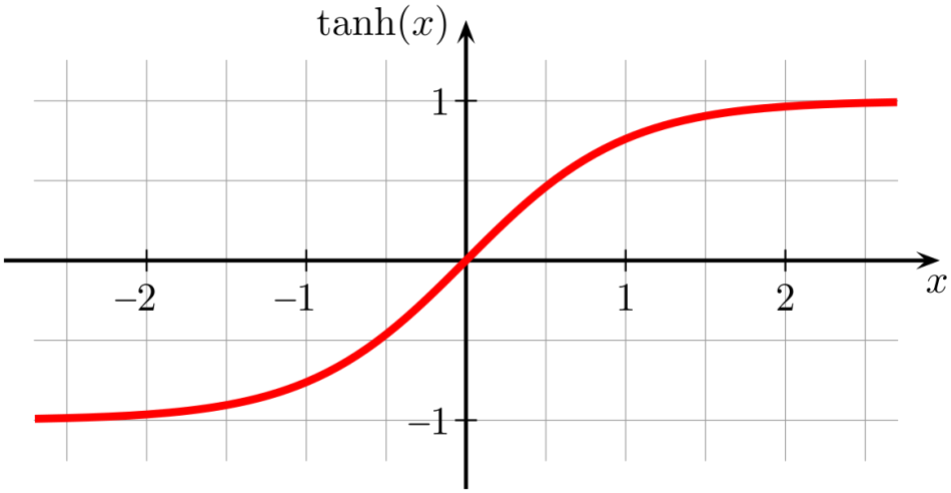

Figura 1.13: Gráfica de la Tangente Hiperbólica. Fuente: ResearchGate.

- **Unidad de rectificación lineal (ReLU).** Esta función de activación sigue un esquema muy sencillo: devolver 0 si se recibe un valor negativo, o devolver el valor si es mayor que 0. Existe una variante de esta función (Leaky ReLU) que no devuelve 0 si el valor es menor que z, sino que

devuelve un valor perteneciente a una recta preconfigurada (véase Figura 1.14).

$$ReLU(z) = \begin{cases} 0 & si \quad z < 0 \\ z & si \quad z \geq 0 \end{cases} \tag{1.9}$$

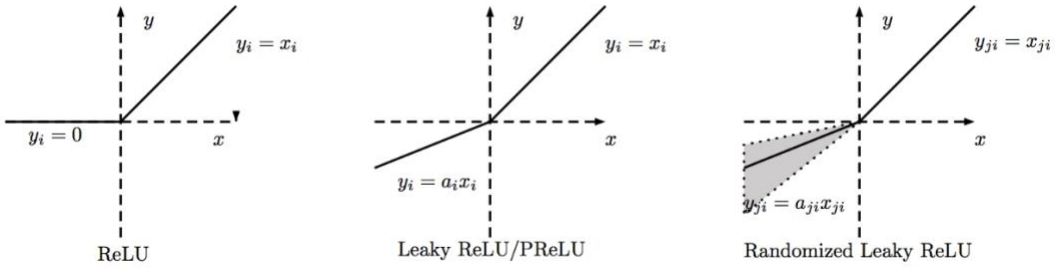

Figura 1.14: Gráfica de la Unidad de Rectificación Lineal (ReLU). Fuente: github.io

- **Lineal.** No aplica ningún operador: se propaga el resultado hacia la siguiente capa, sin modificar.

Por tanto, sirviendo también de aclaración en cuanto a la notación empleada, el procesamiento sería el siguiente.

$z^l = w^{l\,T} * a^{l-1} + b^l$, donde z^l es el valor calculado en la capa L y a^{l-1} es el valor de activación de la capa anterior, computado como sigue: $a^l = g(z)$, donde $g(z)$ es una función de activación de las 4 descritas arriba. En el caso de la primera capa, el valor de activación de la capa anterior, a^0, será el valor de entrada (el ejemplo).

Así, el trabajo de las redes neuronales se asemejará aún más a cómo trabaja un cerebro biológico. Una neurona se activará si la información que recibe en sus dendritas supera cierto umbral, y es esto de lo que se encarga la función de activación.

Tras describir en esta primera parte cómo es la arquitectura de una red neuronal, se da paso a la descripción del entrenamiento de las mismas.

Entrenamiento

La **fase de entrenamiento** es fundamental en cualquier problema relacionado con algoritmos de aprendizaje automático. Consiste en, teniendo el esquema de trabajo del algoritmo y un conjunto de datos de entrada, ajustar los parámetros del algoritmo para generar un modelo que sea capaz de predecir/describir algún tipo de fenómeno mediante el aprendizaje de la distribución probabilística de esos datos. El conjunto de datos de partida ha de dividirse en tres particiones: entrenamiento/desarrollo/test.

El conjunto de **entrenamiento** servirá para entrenar el modelo, mientras que el conjunto de **desarrollo** (o validación) permitirá ir evaluando este entrenamiento y probando la validez del mismo. Una vez que el modelo está perfectamente entrenado, es entonces cuando se utiliza modelo sobre el conjunto de **test**, para probar la precisión del mismo y tener una medida de rendimiento más general.

Según el esquema de aprendizaje o la tarea a resolver, la manera de realizar esto es una u otra específica, dando lugar a diferentes representaciones del conocimiento. En el caso de las **redes neuronales**, el aprendizaje parte de un modelo que ya está construido, compuesto por pesos y sesgos que se han generado aleatoriamente en el intervalo $[0,1]$, y el objetivo es ajustar estos pesos para que el modelo se comporte de la manera en que se espera.

En los siguientes apartados se describen los elementos básicos del entrenamiento de una red neuronal.

Back-propagation

El algoritmo ***back-propagation*** surge a mediados de los años 60 como única solución al hasta entonces no resuelto problema de entrenamiento de redes neuronales profundas. Fue entonces cuando comenzaron a popularizarse de nuevo las redes neuronales desde que el modelo matemático de neurona había sido propuesto (McCulloch & Pitts, 1943). Culminando en 1969 con la publicación en su libro, Minsky y Papert analizaron sus limitaciones (Amari, 1993; Cun, 1988).

Básicamente, *back-propagation* significa “propagación hacia atrás del error”, y es esto precisamente lo que sucede. Este algoritmo toma datos como entrada, los propaga por la red hasta obtener un valor de salida, y luego retrocede hacia la capa de entrada, ajustando los pesos en función del error cometido en la predicción. Si la predicción fue correcta, entonces la medida de error será nula y no variarán los pesos sinápticos. Véase la Figura 1.15 para ilustrar la explicación anterior.

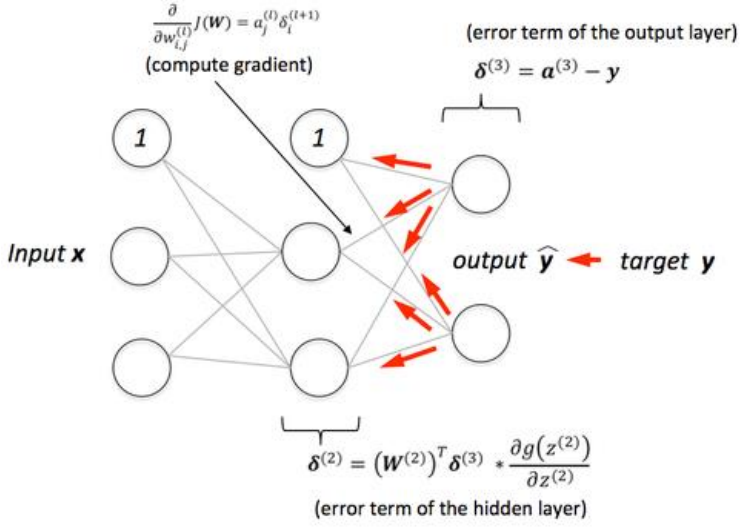

Figura 1.15: Algoritmo Back-propagation. Fuente: sitio web de Sebastian Raschka

La forma en que se recogen los datos de entrada y se hace la actualización de pesos determina la manera en que se hace el entrenamiento de la red. Así, el entrenamiento se define como el problema clásico de optimización, donde existe un espacio de soluciones, y el objetivo encontrar la configuración de parámetros óptima para nuestra red neuronal.

Esquema del algoritmo:

1. Se toma un ejemplo del conjunto de entrenamiento y se propaga hacia adelante hasta obtener una salida.
2. Se compara el valor obtenido con el valor real, y se calcula el coste asociado, obteniendo una métrica de cuál ha sido el error cometido por la red en la predicción.

3. Aplicando el gradiente descendente, se computa la derivada parcial de la función de coste en cada capa con respecto a los pesos que la componen, hasta alcanzar la primera capa.
4. Se actualizan los pesos y sesgos de cada capa según el error cometido y el ratio de aprendizaje.
5. Repetir el proceso varias veces para todos los ejemplos del conjunto de entrenamiento.

De esa forma se aplica el algoritmo *back-propagation* para entrenar una red neuronal. La aplicación de este algoritmo lleva asociados una serie de problemas como, por ejemplo, la suposición de que el resto de capas van a permanecer estáticas al realizar una actualización de pesos en una capa, esto es, se supone el resto de pesos en las capas restantes no variarán. Esto provocará que para encontrar la solución óptima haya que repetir el proceso varias veces y, además, utilizar alguna **heurística de búsqueda** que permita alcanzar el óptimo de la forma más estratégica posible.

Se define el **entrenamiento** como un problema de búsqueda de parámetros en un espacio multidimensional. Cada peso de la red constituye una dimensión del espacio distinta, y el cálculo de las derivadas parciales permite conocer el vector gradiente con la derivada de la función con respecto a cada parámetro, es decir, la dirección que debe seguirse para alcanzar el punto en el que la función de coste se minimiza. Cada punto de esta función de coste representada en el espacio multidimensional constituye una configuración diferente de pesos para la red neuronal. El objetivo es encontrar el punto de la función que representa la configuración de pesos donde el valor de la función de coste es mínimo global.

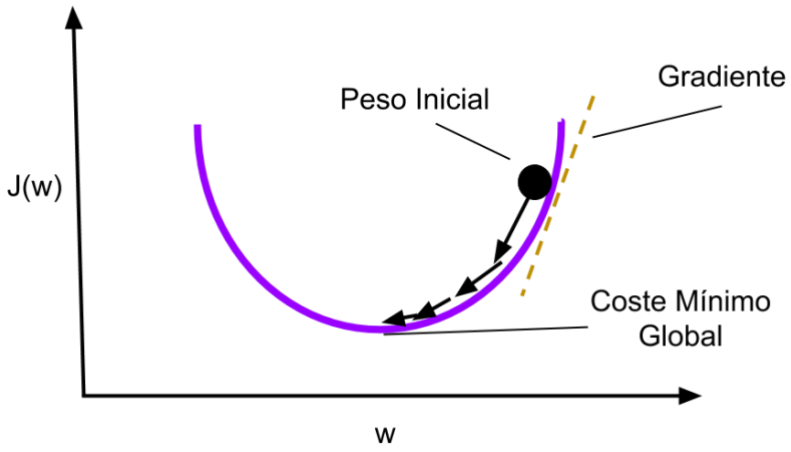

Figura 1.16: Aplicación del Gradiente Descendente. Fuente: Medium

En la imagen superior (véase Figura 1.16) se observa una simplificación del problema original en el que el espacio de pesos se ha representado en una sola dimensión (abscisa), mientras que el valor de la función de coste se ha representado en la ordenada. Como se puede ver, la aplicación del gradiente indica la dirección a seguir para encontrar el óptimo.

Como todo algoritmo de búsqueda, existen los problemas inherentes como son la caída en mínimos locales, la no-convergencia, etcétera. Para solucionar esto, existen una serie de métodos ya propuestos que permiten encontrar una solución cercana a la óptima siguiendo un procedimiento heurístico en la aplicación del algoritmo *back-propagation* (Figura 1.15).

Mediante la construcción de un **grafo de computación**, es posible calcular de una manera rápida y eficiente todas las derivadas en cada capa de la red siguiendo la regla de la cadena. En la Figura 1.17 se observa un ejemplo para el grafo de computación.

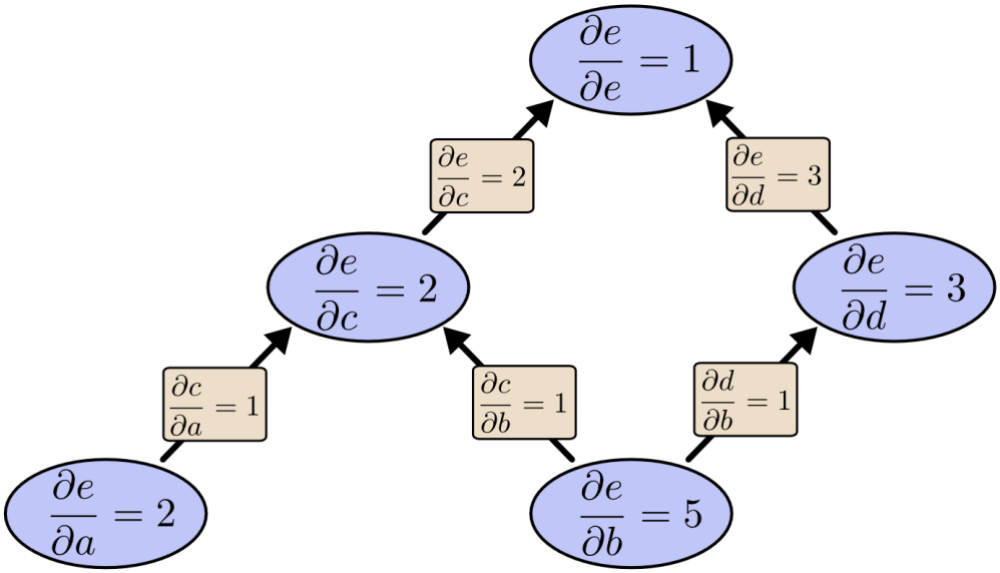

Figura 1.17: Grafo de computación. Fuente: github.io

A continuación, se describen cuáles son los principales métodos de búsqueda que se aplican durante el entrenamiento de redes neuronales.

Optimización

Para resolver el problema del entrenamiento, se plantea como un problema de búsqueda en el que el objetivo es encontrar en un espacio multidimensional en el que cada eje representa a un peso de la red, la configuración óptima que minimiza el coste de la función objetivo (Ruder, 2016). A continuación, se proponen las diferentes alternativas:

Gradiente Descendente Mini-batch

Mediante la **vectorización**, se puede procesar un conjunto de $\boldsymbol{m}$ ejemplos en una sola propagación, lo que hace que el proceso de entrenamiento sea más rápido. En cambio, si m es muy grande, el proceso podría volverse demasiado lento.

Con el modelo clásico, el gradiente descendente no hace nada hasta que no se han procesado todos los ejemplos del *dataset*. Es decir, no actualiza ningún parámetro hasta entonces.

Con este nuevo enfoque, el procedimiento consiste en aplicar el gradiente descendente cada vez que se propaga un *mini-batch*. La ventaja se encuentra en que el gradiente descendiente irá variando gradualmente pues se aplicará más veces, por lo que la función de coste se minimizará más rápido. En cambio, también se darán más saltos por el espacio de soluciones, en pos de una mayor velocidad en el entrenamiento.

Un parámetro a considerar es el tamaño del *batch*. Si:

- **m = número de ejemplos.** Gradiente Descendente Batch. Es ideal, pero es muy ineficiente si el número de ejemplos es grande.
- **m = 1.** Gradiente Descendente Estocástico (SGD, *Stochastic Gradient Descent*) (Amari, 1993). Es eficiente, pero tarda mucho en converger. Se pierde la ventaja de la vectorización, y podría mejorarse con un ratio de aprendizaje menor.
- **m > 1 y m < número de ejemplos.** Es la mejor alternativa. Continúa siendo Gradiente Descendente Estocástico, y reúne lo mejor de alternativas anteriores. Por una parte, el gradiente converge más rápido. Y por otra, no es necesario procesar el dataset entero para conseguir progreso en el entrenamiento.

A continuación, se observa (véase Figura 1.18) la diferencia existente entre aplicar m = número de ejemplos (Gradiente Descendente Batch, a la derecha) y m < número de ejemplos (SGD, a la izquierda).

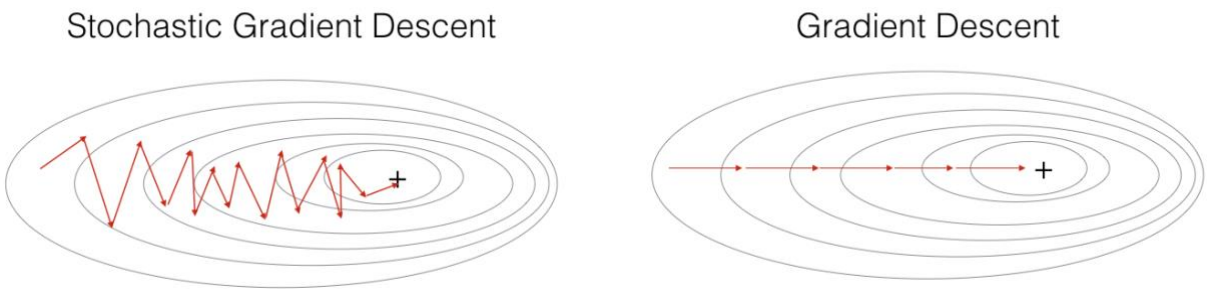

Figura 1.18: Gradiente Descendente Estocástico versus Gradiente Descendente Batch. Fuente: Algorithmic Intelligence Laboratory

En la figura de la izquierda se dan saltos de manera estocástica en la amplitud del espacio de soluciones. En cambio, la exploración del espacio de soluciones es mucho más exhaustiva, y de esta forma es más sencillo encontrar la solución óptima. Mientras que con el Gradiente Descendente Batch se avanza de manera precisa hacia la solución óptima, no consigue alcanzarse por completo, mientras que con SGD, la solución óptima sí es alcanzada.

Gradiente Descendente con *Momentum*

La idea básica de este esquema es calcular la 'media ponderada exponencial' y usarla para actualizar los pesos. Existe un **parámetro de fricción ß**, que va midiendo la velocidad con la que converge el gradiente descendente. En definitiva, **permite escapar de los óptimos locales** (véase Figura 1.19).

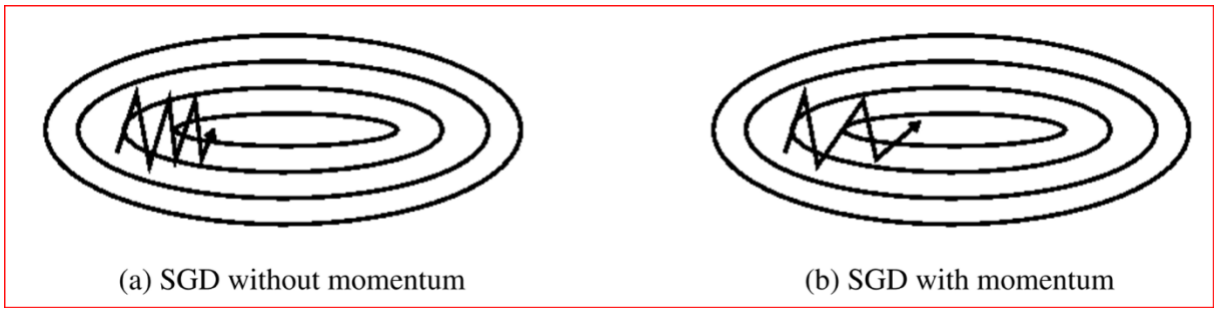

Figura 1.19: Gradiente Descendente con Momentum. Fuente: Medium

RMSProp

RMSProp son las siglas de '*Root Mean Square Propagation'*, o 'Propagación de la raíz cuadrática media'.

Se trata de otro algoritmo de optimización que pretende acelerar la convergencia del gradiente descendente. Sin entrar en detalles matemáticos, veamos este algoritmo como un paradigma que pretende moverse más rápido por el espacio de los pesos *W*, que el espacio de los sesgos *b*.

Adam

Este algoritmo combina lo mejor del Gradiente Descendente con *Momentum* y RMSProp, describiéndose así como una alternativa muy confiable que permite acelerar aún más la convergencia del gradiente descendente (Kingma & Ba, 2014).

Sobreajuste

Los algoritmos basados en redes neuronales presentan una tendencia mayor que el resto de algoritmos de aprendizaje automático al fenómeno que se conoce como **sobreaprendizaje**. Esto es, extraen un conocimiento que resuelve la tarea de manera excelente con los datos de entrenamiento, mientras que, con los datos de validación y posterior test, el resultado empeora significativamente.

En esta clase de algoritmos son necesarias una serie de medidas como utilizar un *dataset* lo más abundante posible, balanceado (el mismo número de casos por cada clase), además de aplicar técnicas de aumento de datos para que el *dataset* pueda aumentar de tamaño y la red sea entrenada con más ejemplos.

Por ejemplo, si se trata de clasificación de imágenes y se dispone de un *dataset* con fotografías de animales, realizar aumento de datos podría ser voltear todas las imágenes respecto al eje vertical, o realizar pequeñas variaciones sobre el color, dando lugar a un mayor número de imágenes.

Para detectar el sobreajuste se sigue una estrategia basada en comparar la variación de la función de coste tanto en entrenamiento como en la validación (véase Figura 3.1). Si en algún *epoch* las gráficas comienzan a separarse una de la otra, de manera que el coste en entrenamiento continúa reduciéndose mientras que el coste en validación comienza a aumentar a partir de ese punto, entonces existe un claro signo de sobreajuste y será necesario reconfigurar los hiperparámetros o bien la red no podrá mejorar el entrenamiento a partir de ese *epoch*.

Hiperparámetros

La cantidad de ajustes que necesita una red neuronal hace que su manejo no sea trivial. De todas las configuraciones de parámetros posibles, solo es posible influir directamente sobre una pequeña parte. Todo lo relativo a ajuste de pesos y sesgos se realiza automáticamente por parte del algoritmo *back-propagation* (Figura 1.15) durante el **entrenamiento**. En cambio, existen ajustes como los **hiperparámetros**, que deben configurarse manualmente antes del entrenamiento de cualquier red neuronal. Algunos de estos hiperparámetros son los siguientes:

- **Ratio de aprendizaje.** Determina en qué medida se va a ver afectado cada peso de la red según el error cometido en cada capa, una vez que se haya hecho la propagación hacia atrás. Si este valor es muy grande, el gradiente descendente dará saltos muy prominentes y rara vez se encontrará una solución aceptable. En cambio, si es muy grande, se tardará mucho en converger a la solución. Lo ideal es moverse en el rango $[10^{-4}, 10^{-2}]$.
- **Número de *epochs*.** Este valor hace referencia a cuantas veces va a procesarse el conjunto de entrenamiento en toda su extensión. Un valor muy elevado puede conducir a sobreajuste.
- **Algoritmo de optimización.** Se tomará esta decisión en base al tamaño del conjunto de entrenamiento y también la complejidad de estos datos. Para datos poco voluminosos y sencillos, se usará el Gradiente Descendente Estocástico con *momentum*. Para datos voluminosos y complejos, podremos elegir RMSProp, o Adam, de manera que se acelere la convergencia hacia la configuración de parámetros óptima.
- **Función de coste.** Según la naturaleza de nuestro problema, se elegirá una función de coste en concreto. Las más comunes son el Error Cuadrático Medio (RMSE, *Root Mean Square Error*, en problemas de regresión), Entropía Cruzada (*Cross Entropy*, para problemas de clasificación), etc.
- **Decaimiento del ratio de aprendizaje.** Una estrategia para evitar el sobreajuste puede ser la de reducir en cada *epoch* el ratio de aprendizaje a una fracción de su valor total. Esta fracción suele ser el 95%, el 90%, o similares. Incluso, podría utilizarse una escala logarítmica.
- **Diseño de la red.** Tal vez, lo más importante sea escoger correctamente el número de unidades ocultas, cuántas capas, cuántas unidades de entrada/salida, etcétera. Esto se suele hacer diseñando algún algoritmo genético que pruebe diferentes configuraciones preestablecidas y seleccione la que aporta el mejor resultado. Es un proceso lento.

Redes Neuronales Convolucionales

Las Redes Neuronales Convolucionales surgieron debido a la gran cantidad de parámetros, la alta dimensionalidad, y el elevado número de conexiones entre la capa de entrada y la capa posterior que se generaba al diseñar una red neuronal para clasificación de imágenes (Goodfellow et al., 2016). La idea era dar como entrada a la red cada píxel de la imagen, por cada uno de los canales que constituyan la imagen. Por tanto, en una imagen de 64 píxeles de ancho * 64 píxeles de alto, suponiendo los canales RGB, se tendrían 64 * 64 * 3 = 12.288 entradas de la red.

Teniendo en cuenta las dimensiones de las imágenes con las que es habitual trabajar hoy en día (hasta 40 megapíxeles, esto es, 40 millones de entradas por cada canal), es fácil darse cuenta de que esa alternativa es inviable. Es en este contexto donde surgen las redes neuronales convolucionales. Su potencia reside en la compartición de parámetros para todos los píxeles de la imagen, con el objeto de extraer características que finalmente servirán como entrada a una red neuronal tipo Perceptrón Multicapa. Para extraer estas características se usa la operación matemática de **convolución**, si bien en un sentido diferente a la convolución tradicional para dos funciones.

Principalmente, en este tipo de redes se distinguen capas de 3 tipos: **convolución**, **reducción**, y ***fully-connected***. Se describirán las dos primeras, puesto que la última no deja de ser un perceptrón multicapa que recibe como entrada un conjunto de características extraído en fases previas mediante la convolución.

Capas de convolución

En la **convolución** se realizan operaciones de productos y sumas entre la capa de partida y los n filtros (o *kernels*), generando un mapa de **características** (Szegedy et al., 2015). Las características extraídas corresponden a cada posible ubicación del filtro dentro de la imagen original (véase Figura 1.20).

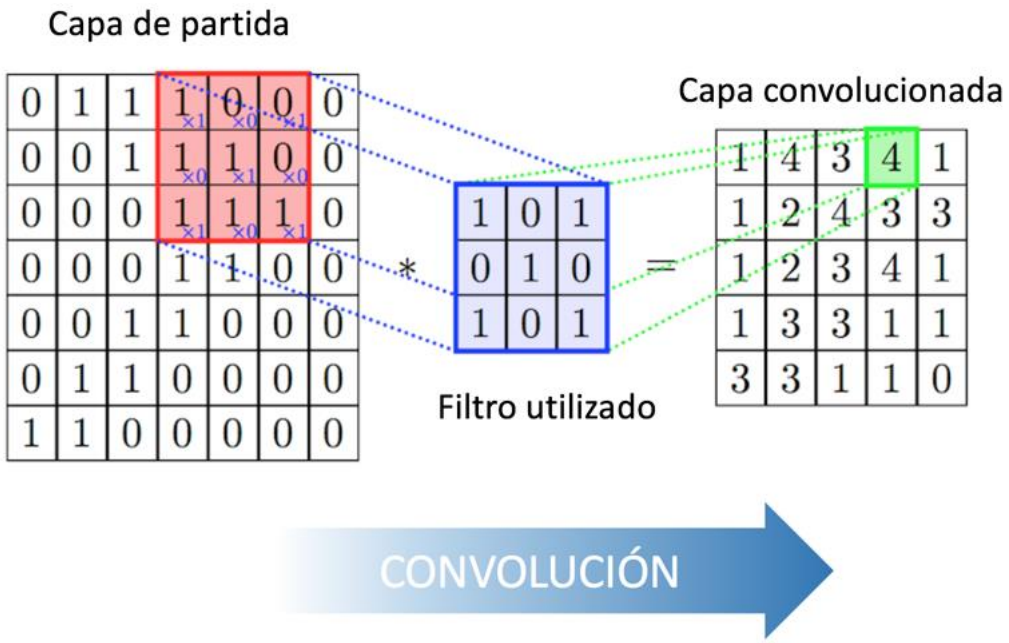

Figura 1.20: Operación de convolución. Fuente: sitio web de Diego Calvo

La ventaja es que el mismo **filtro** sirve para extraer la misma característica en cualquier parte de la imagen, y con esto se consigue reducir el número de conexiones y el número de parámetros a entrenar en comparación a una red neuronal tipo perceptrón multicapa.

Después de aplicar la convolución, se aplica sobre estos mapas de características una función de activación: una de las descritas en la sección anterior.

Los filtros de la capa de convolución se encargarán de extraer características sobre la entrada actual, para lo cual se realizan operaciones de productos y sumas como ya se ha visto. En cambio, los pesos que componen un filtro no se ajustan de forma manual y fija, sino que se inicializan aleatoriamente, y se entrenarán utilizando el algoritmo *back-propagation* (Figura 1.15) de la misma forma en que se entrena el resto de capas de la red, de manera que cada filtro aprenda a extraer una característica en concreto.

En cada capa se aplicará un número arbitrario (preestablecido) filtros, de los cuáles, cada uno será responsable de extraer una **característica** diferente sobre la

imagen de entrada. La complejidad de las características tiene un carácter jerárquico, y se incrementa con la profundidad en la red.

En una capa de **convolución** existen una serie de hiperparámetros que han de configurarse, como por ejemplo el *stride*, que indica cuántos píxeles se avanza tanto en horizontal como en vertical (por defecto se usa un *stride* de valor 1). Un problema que tienen estas redes es que se suelen perder muchas características de regiones cercanas a los bordes de la imagen. Por tanto, surge otro parámetro, el *padding*, que indica el desplazamiento a aplicar sobre los datos de entrada. Lo que hace es suponer que la imagen o el mapa de características de entrada tiene un tamaño mayor al tamaño real, pasando más veces por los bordes de la imagen y extrayendo también esas características. Cada capa de convolución tiende a disminuir la dimensionalidad de la imagen de entrada.

Este tipo de capa determina qué características y en qué cantidad son relevantes para el tipo de problema en el que se aplican, dotando de mayor o menor expresividad en función de cómo se configuren.

Capas de reducción

En la reducción o *pooling* se disminuye la dimensionalidad de la entrada quedándose con las características más comunes. La forma de reducir la dimensionalidad es utilizar estadísticos como la **media** o el **máximo** de un conjunto de celdas. Al reducir se pierde precisión, pero se gana en compatibilidad. Haciendo que la red pueda clasificar mejor, aunque haya ejemplos de entrada que no ha visto en su totalidad.

En esta capa se mantienen los conceptos de la capa anterior, salvo que no hay un filtro que tenga que entrenarse. De hecho, en la literatura se observa una gran discrepancia con respecto a considerar las capas de reducción como un tipo de capa más, o meramente son un operador matemático, puesto que en esta capa no se realiza ajuste de pesos. El filtro, en lugar de multiplicarse por la entrada, se limita a quedarse con el máximo o la media del conjunto de píxeles encerrados por el mismo. Por tanto, existen dos tipos de reducción: ***Max Pooling*** **y** ***Average Pooling***. La Figura 1.21 ilustra el funcionamiento de ambos.

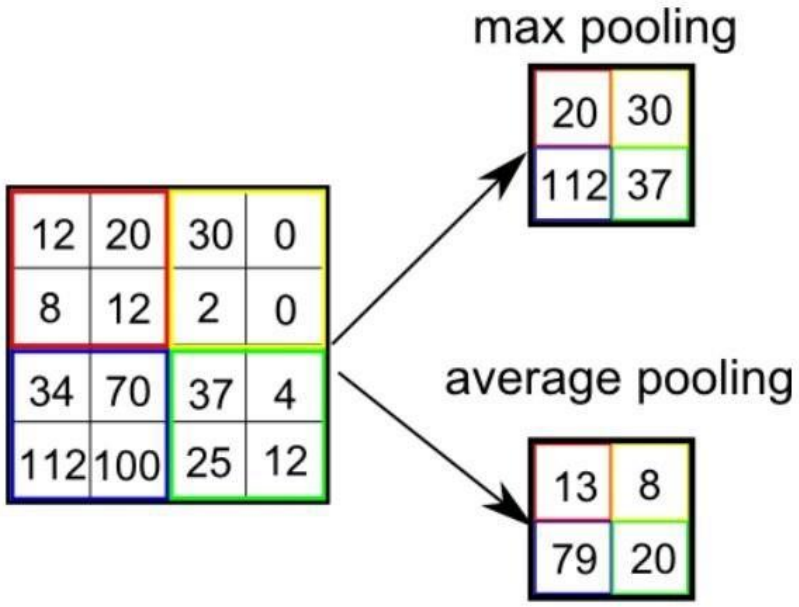

Figura 1.21: Max Pooling versus Average Pooling. Fuente: Quora

Capas *fully-connected*

Aunque en la literatura se conoce a estas capas como *fully-connected*, no deja de ser un conjunto de capas de una red neuronal tipo **perceptrón multicapa** (Figura 1.4). La diferencia con respecto a la versión clásica de alimentar la red directamente con los píxeles de la imagen reside en que ahora la red se alimenta con características, con estructuras que ya poseen carga semántica, habiéndose descartado todo el contenido no relevante de la imagen.

Al recibir características como entrada, este tipo de red también permite reconocer entidades con independencia de la localización en la imagen. Por ejemplo, a detectar una cara independientemente si está en la parte superior de la imagen o en la parte inferior.

Naturaleza jerárquica

Estas redes trabajan de un modo **jerárquico**. Esto es, parten de las características más simples, construyendo estructuras cada vez más complejas, hasta conseguir características de alto nivel con las que servir de entrada a la última parte de la red.

Esto es algo característico del aprendizaje profundo. En la primera capa de convolución suelen extraerse características muy básicas como bordes, esquinas, etcétera. En las siguientes capas se tiende a extraer formas, resultado de componer características extraídas en capas anteriores. De esta manera, se realiza un procesamiento incremental en el que cada capa posterior maneja características de más alto nivel. El ajuste de todas las capas se realiza automáticamente, extrayendo las características con las que mejor rendimiento se obtiene.

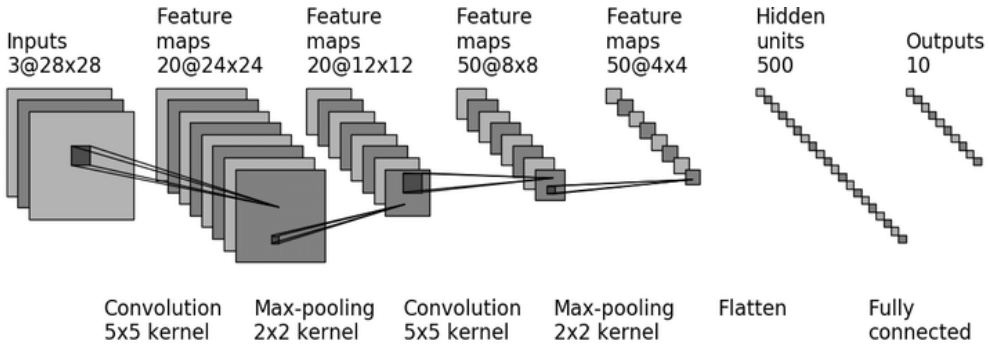

Figura 1.22: LeNet-5. Fuente: ResearchGate

Siguiendo esta filosofía, existen una serie de arquitecturas propuestas que han tenido bastante repercusión científica en tareas específicas:

- **LeNet-5.** El objetivo de esta arquitectura (Figura 1.22) era reconocer los caracteres manuscritos del *dataset* MNIST en escala de grises (Bengio, LeCun, & Henderson, 1993).
- **AlexNet.** Esta arquitectura fue la ganadora en 2012 del reto ImageNet de reconocimiento de imagen (Krizhevsky et al., 2012). Tomaba como entrada imágenes RGB de 227 * 227.
- **VGG-16.** Este tipo de red fue posterior a AlexNet y simplificó mucho las arquitecturas de redes neuronales, centrándose sobre todo en la convolución (Simonyan & Zisserman, 2015).

Redes Neuronales Convolucionales 3D

Como último paso a dar antes de exponer todo lo relativo a gráficos, nubes de puntos, etcétera, ha de concluirse este bloque dando a conocer las redes neuronales convolucionales 3D, objeto de sumo interés para el presente Trabajo Fin de Grado.

Siguen exactamente la misma filosofía descrita hasta ahora, con la diferencia de que en lugar de trabajar sobre imágenes 2D, lo hacen sobre volúmenes 3D. Normalmente, *grids* de vóxeles, como será el caso.

Todas las operaciones se hacen de la misma forma, pero añadiendo una nueva dimensión, la profundidad (o eje z). En el caso de las capas *fully-connected*, este *grid* de características resultante se desplegará en un vector de características extendiéndose solamente en una dimensión, con el que se alimentará esta capa.

Inicialmente, esta morfología de capa se ideó para la extracción de características en vídeo, pues se componían todos los fotogramas formando un volumen, del que luego se irían extrayendo características. Nuestro enfoque pretende ir un paso más allá y utilizar esta arquitectura para extraer características de *grids* de vóxeles, aplicándola en segmentación/clasificación de nubes de puntos. En secciones posteriores se describirá el proceso de cómo se ha utilizado una arquitectura de red que utiliza capas de convolución y reducción 3D para resolver la tarea de la **segmentación de nubes de puntos**.

1.3.3 Representando el mundo

Hasta la fecha, el tratamiento de datos espaciales en cuanto a segmentación/detección de objetos ha sido objeto de estudio por numerosos grupos de investigación alrededor del mundo. Si bien representar el mundo real en un ordenador es una tarea extremadamente compleja por la abundancia de información, se han conseguido avances muy prometedores.

Cuando de representar el mundo se trata, se dispone de dos enfoques principales: usar 2 dimensiones, capturando una instantánea de la realidad en forma de imagen, o extraer una representación literal usando directamente las 3 dimensiones.

1.3.3.1 Dos dimensiones

Representación de la información

Representar la realidad usando únicamente dos dimensiones, hace necesario tener que proyectar toda la información sobre un plano, perdiendo muchos aspectos relevantes como la profundidad. Por tanto, la representación de la realidad se da de una forma muy sesgada, dificultando cualquier tratamiento de la información que necesite más datos además de los que podrían verse en una imagen.

Principalmente, para representar datos bidimensionales se utiliza la imagen en sus diferentes formatos (JPEG, PNG, BMP, etcétera). Una imagen suele ser normalmente capturada utilizando una cámara fotográfica, pero también es posible generar imágenes sintéticas en computadores utilizando un FBO (*Framebuffer Object*) mediante el ***rendering*** de una escena.

Normalmente, una imagen está formada por 1 solo canal (escala de grises) o 3 canales (RGB), pero no están limitadas solo a esto, pudiendo contener mucha más información como es el caso de las imágenes **multiespectrales**, como la que se visualiza en Figura 1.23**.**

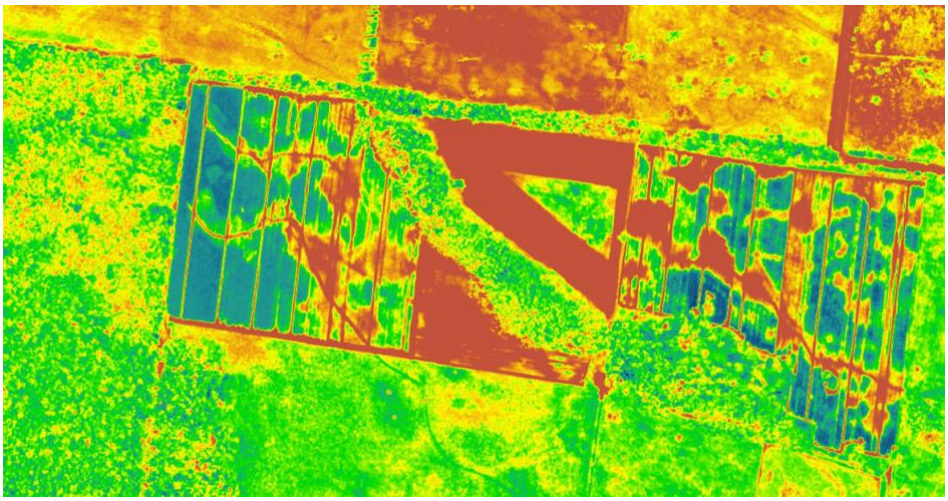

Figura 1.23: Imagen multiespectral. Fuente: Google

Segmentación en datos 2D

La segmentación de objetos en imágenes 2D tendría como resultado determinar qué píxeles de la imagen son distinguibles del resto debido a que comparten propiedades entre sí.

El hecho de poder distinguir cuántos objetos hay en una imagen o dónde están o incluso qué objetos son si lo abordamos desde la clasificación, de manera automática mediante el uso de un ordenador, resulta de interés en algunos campos como la conducción autónoma, o elaboración automática del inventario de árboles en una finca, entre otras aplicaciones. Este tipo de datos se recogerían mediante cámaras en el vehículo o el vuelo de un dron, respectivamente, y servirían de entrada a un tipo de algoritmos especializados.

Existen técnicas clásicas de segmentación en imágenes que fueron precursoras en el área. De hecho, la segmentación, propiamente dicha, es un concepto que viene de la mano del procesamiento de imágenes, definido así: **"El resultado de la segmentación de una imagen es un conjunto de segmentos que cubren en conjunto a toda la imagen, o un conjunto de las curvas de nivel extraídas de la imagen. Cada uno de los píxeles de una región son similares en alguna característica, como el color, la intensidad, o la textura. Regiones adyacentes son significativamente diferentes con respecto a la(s) misma(s) característica(s)."** Sin embargo, el auge del aprendizaje automático ha impulsado

nuevas técnicas que dejan a un lado a las técnicas tradicionales para abordar el problema con un nuevo paradigma que pretende resolver la tarea mediante la extracción automática de características. La resolución de este tipo de problemas es muy sencilla para los humanos, y muy compleja para un ordenador debido a la dificultad de especificar formalmente qué aspecto tiene una característica.

Para entrenar un algoritmo de aprendizaje automático que sea capaz de segmentar objetos en imágenes se necesita adquirir un volumen significativo de imágenes etiquetadas con los objetos que contienen y las coordenadas relativas en píxeles dentro de la imagen.

Cuando se realiza un cambio de contexto pasando del plano en 2D al espacio en 3D, todo se dificulta. El hecho de añadir una nueva dimensión no solamente introduce una dificultad a la hora de procesar la información, siendo más voluminosa, sino que además la acción de escoger una representación ya es en sí un reto, al no haber una única alternativa como es el caso del 2D, en el que con frecuencia se utilizan imágenes para representar la información.

1.3.3.2 Tres dimensiones

Cuando se añade una dimensión más, el paradigma de representación cambia, abriéndose un abanico de posibilidades para representar la información tridimensional. Este extra de información tiene una complejidad inherente, no solo en volumen, que se ve aumentado pues cada coordenada bidimensional ha de multiplicarse por una tercera, sino también por la dificultad de su procesamiento al no existir ninguna forma estándar para representar este tipo de información.

A continuación, se van a describir todas las formas de representación 3D que se utilizan: mallas, volúmenes, imágenes y nubes de puntos; con especial interés en esta última, objeto principal de estudio en este proyecto.

Tipos de representación

En el manejo de información tridimensional se dispone de cuatro tipos básicos de representación, y la elección de una en particular viene condicionada por el

objetivo que se pretende conseguir, o simplemente por el formato de salida del dispositivo de captación de esta información. Principalmente, se distingue entre:

Mallas

Se trata de una **colección de polígonos y vértices** que aproximan una superficie 3D. Se define un conjunto de vértices, y por otra parte se definen las primitivas (mediante índices a los vértices, pueden ser triángulos o quads) que especifican cómo se conectan esos vértices (véase Figura 1.24. Es uno de los tipos de representación más utilizado para información tridimensional por su carácter de precisión incremental. Se puede aproximar cualquier tipo de superficie puesto que la tecnología lo permite, sin incrementar mucho el volumen de información. Además, distintas regiones pueden tener precisiones diferentes, siendo este un aspecto favorable porque reduce la complejidad de la información.

Con una malla se tiene habitualmente una **representación 2,5D** de los objetos, es decir, solo se tienen detalles sobre la superficie, mientras que todo lo que hay debajo (o dentro) es desconocido. Así, se dispone de una especie de representación volumétrica ficticia, pues en realidad no existe una figura sólida, sino una serie de planos interconectados que gracias a su topología consiguen dar un aspecto de volumen.

Este tipo de representación es el más utilizado en todas las aplicaciones de informática gráfica, desde videojuegos hasta programas de CAD/CAM (*Computer Aided Design / Computer Aided Modelling*) o BIM (*Building Information Modeling*).

Por su **variabilidad e irregularidad** no se puede considerar este enfoque para diseñar una red neuronal que acepte como entrada directamente mallas, debido a que cada malla podría tener un número de vértices diferente, o el mismo pero distinto número de primitivas. Además, la red no solo tendría que recibir como entrada una malla, sino que tendría que consumir una escena completa, y sobre ella segmentar.

La tendencia a segmentar el espacio automáticamente tiene el propósito de tener una mejor visión de la realidad, así como diseñar sistemas que trabajen de

manera autónoma en el mundo real. Por la complejidad inherente, resulta muy difícil aproximar una malla a cada objeto de la realidad. La recursión tendería a infinito y habría que definir un nivel de precisión. Por tanto, esta representación suele utilizarse meramente en representaciones que no van más allá de los videojuegos, el CAD/CAM, etcétera. Los algoritmos de generación de mallas a partir de la realidad son muy complejos, y requieren primero un conjunto de vértices muestreado sobre el mundo real.

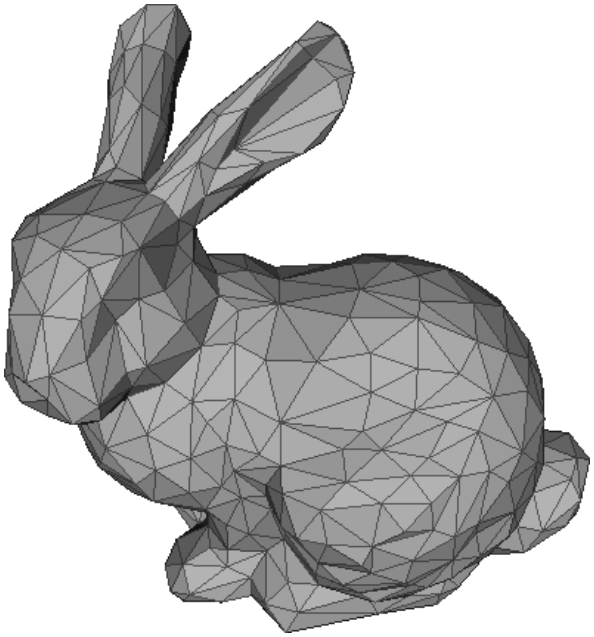

Figura 1.24: Malla de triángulos. Fuente: Stanford

Volumétrica

La representación **volumétrica** presenta una gran similitud con la de mallas. La diferencia es que ahora se tiene una representación 3D en lugar de 2,5D y además desaparece el conjunto de vértices y primitivas. Se tiene la capacidad de representar el volumen de estos objetos de una manera realista utilizando asociaciones de cubos con una precisión preestablecida (véase Figura 1.25).

Se trata de una representación muy sencilla, pues básicamente consiste en un *array* **tridimensional** (*grid*), donde cada una de las celdas corresponde a un elemento que conocemos como vóxel (píxel volumétrico). Un **vóxel** la extensión tridimensional del píxel. En cada celda del *grid* se almacena un valor, indicando si

está presente ese vóxel o no. Aunque existen diferentes modelos, el más sencillo es el *grid* de **ocupación**, donde en cada celda aparece 0 o 1 según si hay un vóxel presente en ella o no. Todos los vóxeles que componen el *grid* han de tener el mismo tamaño $X \times Y \times Z$, así, una de las consideraciones a tratar al optar por esta representación es la **precisión** a utilizar al realizar la transformación desde otra representación. A mayor precisión, más pequeños serán los vóxeles y mayor será la dimensión del *grid*, haciendo también más densa la información.

No existe un formato abierto estándar para almacenar los vóxeles (como podría ser el formato *.pdf* en documentos digitales), pues la representación es tan sencilla que ninguna compañía u organización han apostado por desarrollar un estándar.

La apuesta por la **sencillez** de esta representación provoca una pérdida proporcional de precisión. Es necesario definir una precisión para el vóxel, p. Cada vóxel encerrará una porción del espacio de dimensiones $p \times p \times p$ unidades. Aunque p no estará limitado a ser el mismo valor para las tres dimensiones del espacio, habitualmente sí será el mismo valor. Toda variación espacial dentro del vóxel se pierde de manera que, si una figura es compleja y de dimensiones menores a la precisión del vóxel, se perderá todo detalle pasando a ser representada por un único cubo. Por ejemplo: si la precisión es de 0,2 metros en las tres dimensiones, cada vóxel representará un volumen de 0,008 m^3 del espacio, y toda variación dentro se perderá.

Por esto es importante escoger una precisión. Si por ejemplo se van a representar coches, quizá con una precisión de $[0.15, 0.2]$ metros es suficiente. En cambio, si la realidad a representar son los objetos de una caja de herramientas, es probable que todas las herramientas sean representadas como un único cubo, siendo imposible distinguir detalles, ya que **toda variación dentro de un vóxel se pierde**.

En cambio, un aspecto a favor de esta representación es la **regularidad y uniformidad** mostrada. Al ser todos los vóxeles del mismo tamaño, es posible

establecer una dimensión fija para el *grid* y usar éste para entrenar una red neuronal convolucional con una configuración predeterminada.

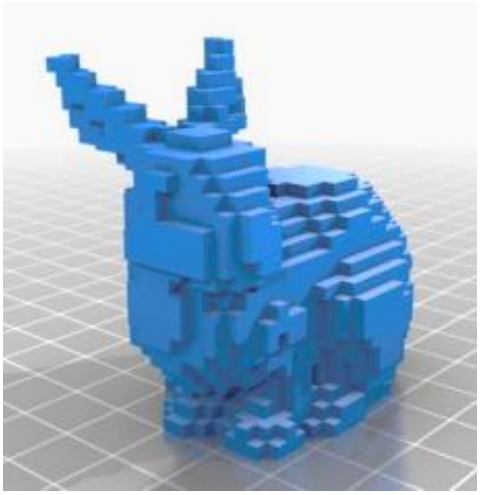

Figura 1.25: Modelo de vóxeles. Fuente: Stanford

La **generación de modelos volumétricos** no suele hacerse captando directamente información del mundo real y presentándola en un ordenador, sino que más bien se trata de una representación útil que suele usarse a partir de la transformación de otros tipos de representación. En el caso de este trabajo, por ejemplo, se utilizarán los modelos de vóxeles para transformar la nube a una representación regular volumétrica.

Imagen RGB (D)

La **imagen** RGB (D) (véase Figura 1.26) es otro tipo de representación para modelos tridimensionales; además de los 3 canales para el color (RGB) se incluye un canal adicional para la profundidad (D) que representa la distancia real del objeto al sensor de captación.

Estas imágenes pueden tomarse usando una cámara fotográfica, o bien ser producto de un proceso de *rendering* en un ordenador. Normalmente, una imagen que sea renderizada por un ordenador no va a ser objeto de un proceso de segmentación, ya que no se necesita conocer qué objetos aparecen en la imagen pues ya se conoce a priori. En cambio, estas imágenes podrían utilizarse como estrategia de aumento de *dataset*, para entrenar algoritmos con formas que son difícilmente captables en la realidad.

La representación mediante imágenes RGB (D), aunque es sencilla, no suele utilizarse debido a que se pierde mucha información tridimensional de gran validez al tener que proyectar la escena sobre un plano bidimensional. Una vez más, la apuesta por la sencillez implica una pérdida de precisión e información importante de carácter tridimensional.

Figura 1.26: Imagen RGB. Fuente: Stanford

Nube de puntos

Una **nube de puntos** constituye un conjunto de datos formado por puntos ubicados en un sistema de coordenadas tridimensional (véase Figura 1.27). Estos puntos son definidos por coordenadas (x, y, z) y están destinados a representar la superficie externa de un objeto. Por tanto, se asemejan a las mallas al ser una representación 2,5D, pero una información menos compleja ya que no existe una topología entre los diferentes puntos. Habitualmente, cada punto contendrá información adicional a la localización (x, y, z), como podrá ser el color RGB, la intensidad, o incluso una etiqueta de a qué objeto o superficie pertenece ese punto.

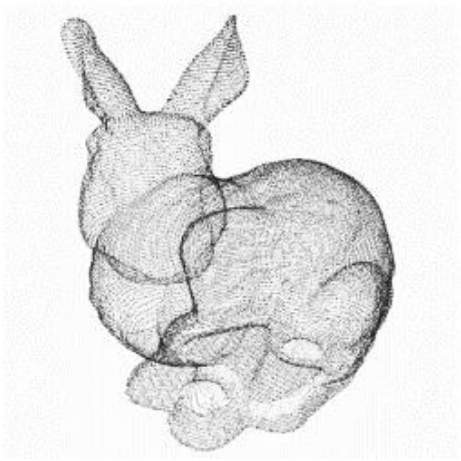

Figura 1.27: Nube de puntos. Fuente: Stanford

Las nubes de puntos son creadas por escáneres 3D que muestrean un gran número de puntos en la superficie de un objeto y producen un archivo de datos que representa el conjunto muestreado por el dispositivo. Si bien, en una sección posterior se describirá que no siempre se utiliza un escáner para obtener nubes de puntos. El archivo resultante contiene toda la información métrica de la superficie escaneada, esto es para cada punto se tienen sus coordenadas (x, y, z) u otro tipo de coordenadas según el sistema de georreferenciación empleado; y a veces metadatos relativos al color y la **reflectividad** del material.

Se utilizan para crear mallas 3D y otros modelos utilizados en el modelado 3D para varios campos, incluyendo imagen médica, arquitectura, impresión 3D, realidad virtual, entre otros. La generación de este tipo de datos suele ser de bajo coste, siendo de gran facilidad su adquisición. En cambio, se trata de datos voluminosos por la cantidad de puntos que pueden llegar a estar contenidos en una nube (orden de varios cientos de millones).

Los objetos descritos por nubes de puntos pueden tener dimensiones de unos pocos milímetros, o tan grande como ciudades completas incluyendo edificios, carreteras, árboles, coches, etcétera, como por ejemplo la nube que se visualiza en la Figura 1.28, correspondiente a un escaneo sobre la ciudad de Jaén. Además de la información para la coordenada de cada punto es habitual incluir el color, dando lugar a representaciones muy realistas.

Figura 1.28: Nube de puntos de la ciudad de Jaén

Para generación de datos de nubes de puntos se presentan diferentes alternativas:

- **Muestreo en CAD/CAM.** A partir de una malla (véase Figura 1.29) como la que se describió en secciones anteriores es posible generar una nube de puntos mediante el muestreo de puntos que pertenecen a polígonos que están en la malla, tal y como se observa en la Figura 1.30). Como ya se ha dicho, una malla es un conjunto de vértices con una relación topológica en forma de polígonos con la que se representa una superficie. Si es posible generar puntos que están contenidos en estos polígonos, se obtiene una nube de puntos. Este procedimiento puede realizarse de una manera muy sencilla utilizando la herramienta **Cloud Compare** (https://www.danielgm.net/cc/)**.** En (Wu, Wan, Yue, & Keutzer, 2018) se opta por esta estrategia al utilizar el videojuego *Grand Theft Auto 5* para generar muestreos de nubes de puntos virtuales con los que después entrenar un algoritmo de segmentación.

Figura 1.29: Modelo CAD de una mesa

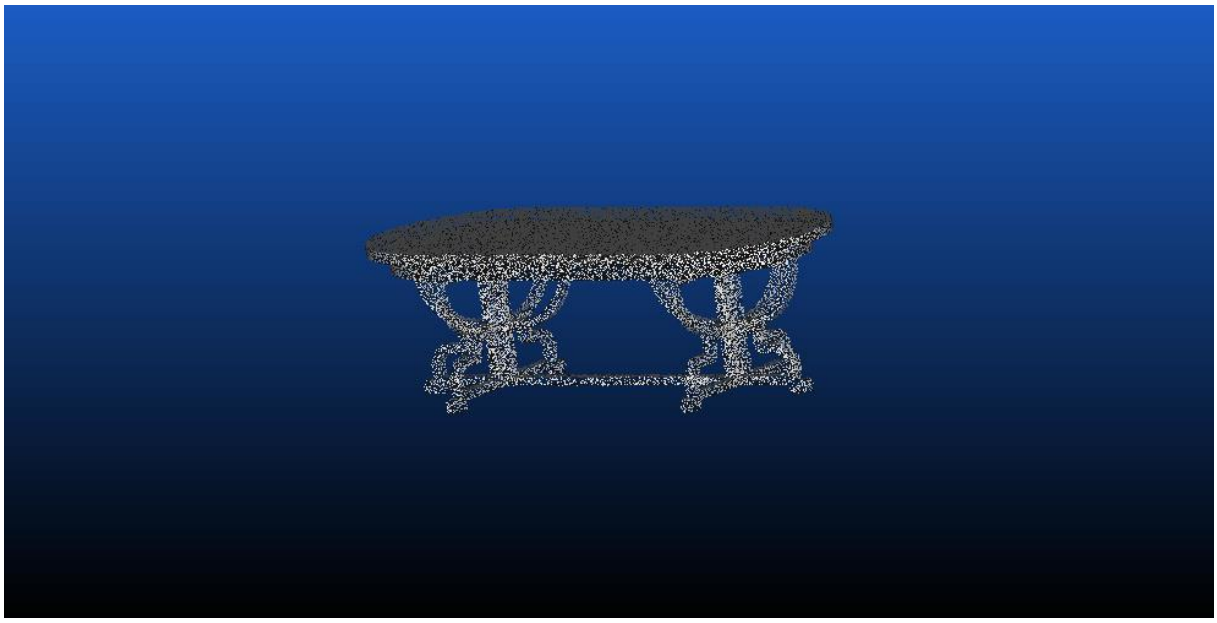

Figura 1.30: Nube de puntos generada mediante muestreo CAD para una mesa

- **SFM (*Structure from motion*).** Se trata de una técnica de bajo coste que tiene su origen en la visión por computador, usada para la obtención de datos de alta resolución, capaz de construir modelos 3D a partir de fotografías 2D del objeto en cuestión (véase Figura 1.31). Este tipo de adquisición, a menudo, tiende a un alto coste debido a que la recogida y captura de los datos puede ser muy complicada por la lejanía o inaccesibilidad en muchos lugares de campo. Otras plataformas como el

escaneo láser (siguiente método) son más baratas, pero menos fiables. El método parte de una serie de imágenes de un objeto en cuestión y la posición desde la que se toman estas imágenes. La etapa de procesamiento inicial es la identificación de características en imágenes individuales que pueden ser usados para relacionarlas entre ellas mediante características comunes. Una de las soluciones propuestas es el **sistema de extracción de características SIFT**: este algoritmo identifica características en cada imagen que son invariantes en la escala, la rotación y los cambios en condiciones de iluminación. De esta manera, los puntos de interés, o 'puntos clave', se identificarán automáticamente todas las escalas y localizaciones en cada imagen, y servirán para encajar todas las imágenes más adelante. Se requiere un mínimo de **3 fotografías** para que el sistema se comporte correctamente. Una vez conseguidos estos puntos se pueden muestrear puntos entre ellos para generar nubes de puntos. La técnica ha sido muy utilizada en proyectos de ingeniería, por ejemplo, para extraer nubes de puntos en mecánica de rocas con el objeto de determinar la orientación de las discontinuidades.

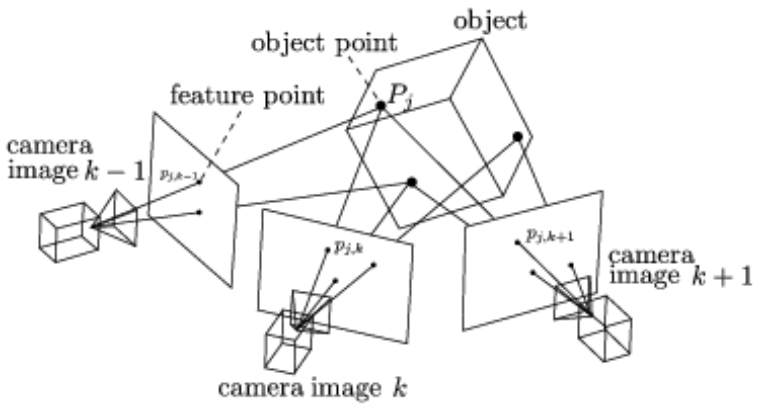

Figura 1.31: Generación de modelos 3D mediante SFM. Fuente: ResearchGate.

- **LiDAR (*Light Detection And Ranging*).** Se trata de un dispositivo que permite determinar la distancia entre un emisor láser a un objeto o superficie utilizando un haz láser pulsado. La distancia al objeto se determina midiendo el tiempo de retraso entre la emisión del pulso y su detección a través de la señal reflejada. Esta tecnología está siendo estudiada para el vehículo autónomo (especialmente, como medio para

capturar el entorno y segmentarlo). Un LiDAR es un sistema que permite obtener una nube de puntos del terreno tomando muestras de puntos mediante un escáner láser aéreo (también está la opción terrestre, TLS). Para realizar el escaneo se combinan dos movimientos: **longitudinal**, dado por la trayectoria del avión, y **transversal**, mediante un espejo móvil que desvía el haz de luz láser emitido por el escáner. Para conocer las coordenadas de la nube (registrar la nube) se necesita la posición del sensor y el ángulo del espejo en cada momento. Para ello el sistema utiliza un GPS diferencial y un sensor inercial de navegación (INS). Conocidos estos datos, y la distancia sensor-terreno obtenida con el distanciómetro (medidor de distancias), se obtienen las coordenadas buscadas. El resultado es decenas de miles de puntos por segundo. Principalmente existen dos tipos de LiDAR: ALS (Escáner Láser Aerotransportado, como el observado en la Figura 1.32) y TLS (Escáner Láser Terrestre).

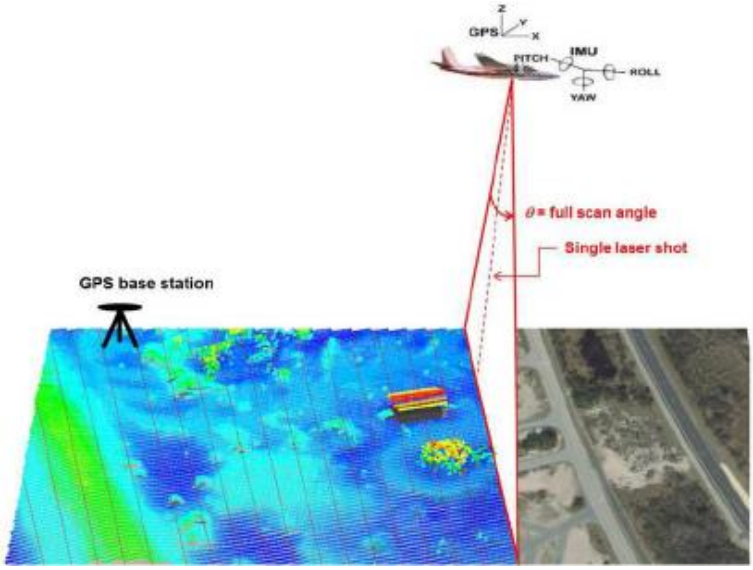

Figura 1.32: LiDAR aéreo (ALS). Fuente: Google.

1.3.4 Segmentación en nubes de puntos

Es el proceso de clasificar nubes de puntos en múltiples regiones homogéneas, donde los puntos en la misma región tendrán las mismas propiedades intrínsecas, o bien comparten una categoría semántica. Se trata de un problema complejo, debido a la alta redundancia y falta de estructura explícita en la nube de puntos.

El método general es construir un grafo para la nube de entrada, generando una malla, y agrupar este grafo para producir una segmentación usando información como la dirección de la normal, suavidad, concavidad, etcétera.

Dragomir Anguelov, un investigador especializado en visión artificial y el procesamiento de nubes de puntos, sugiere que un algoritmo de segmentación de nubes de puntos debe tener tres propiedades importantes (Nguyen & Le, 2013):

- El algoritmo **debería beneficiarse las características cualitativas**, distinguir entre coches y árboles (por ejemplo). Cuando el número de características crece, el algoritmo debería aprender a compensar eso automáticamente.
- El algoritmo de segmentación **debería ser capaz de inferir las etiquetas** de los puntos que pertenecen a regiones muestreadas basándose en la información de sus vecinos.
- El algoritmo de segmentación **debería adaptarse particularmente al escáner 3D utilizado**, dado que los diferentes tipos de escáneres producen nubes de puntos cualitativamente diferentes para la misma escena.

1.3.4.1 Retos

Es posible determinar la forma, tamaño y otras propiedades de objetos en datos 3D. Las **nubes de puntos son habitualmente ruidosas, dispersas y desorganizadas**. No existe un orden natural entre los puntos, de manera que no puede definirse una relación de orden en el conjunto de puntos que forma la nube.

Un punto carece de significado, y necesita de sus vecinos para la creación de estructuras complejas y aumentar su carga semántica.

La **densidad de puntos también es desigual**, estando determinada por la inclinación y la variación del escáner. También, la forma de la superficie puede ser arbitraria con características puntiagudas, no habiendo patrones de distribuciones estadísticas en los datos. Además, debido a las limitaciones de los sensores 3D, lo que está en el primer plano es habitualmente confundido con lo que está en un segundo plano.

1.3.4.2 Técnicas clásicas de segmentación

La tarea de segmentación ha sido ampliamente estudiada por numerosos grupos de investigación alrededor del mundo. Existen una serie de técnicas clásicas (Nguyen & Le, 2013) que han sido el estado del arte en la tarea durante algunos años, y se basan en estimaciones como los vectores normales, el color, etcétera. A continuación, se describen estas técnicas.

Métodos basados en borde

Los bordes describen características sobre la forma de los objetos. Detectan los límites de varias regiones en las nubes de puntos para obtener regiones segmentadas. El principio de estos métodos es localizar los puntos que tienen un cambio rápido en la intensidad. **Bhanu et al.** (Nguyen & Le, 2013) propone una técnica de detección de bordes que calcula el gradiente, ajustando líneas 3D a un conjunto de puntos y detectando cambios en la dirección de los vectores normales unitarios en la superficie. Jiang (Nguyen & Le, 2013) presentó un método rápido de segmentación usando una **técnica de agrupamiento de líneas de escaneo.** Las líneas de escaneo de la **imagen de rango** son divididas en curvas, y luego agrupadas para representar superficies.

Estos métodos permiten segmentación rápida, pero tienen **problemas de precisión debido a que son muy sensibles al ruido y a la densidad desigual de puntos en la nube**, situaciones muy comunes en las nubes de puntos.

Métodos basados en región

Los métodos basados en región utilizan información de sus vecinos para combinar puntos cercanos que tienen propiedades similares para obtener regiones aisladas y, por consiguiente, encontrar diferencias entre las diferentes regiones. Son más tolerantes al ruido, pero tienen problemas de precisión al encontrar el borde. Se distinguen dos enfoques:

- ***Seeded región* (enfoque abajo-arriba).** Comienzan la segmentación eligiendo algunos puntos semilla, y luego, desde esos puntos, cada región podrá crecer añadiendo puntos vecinos si satisfacen cierto criterio o límites de comparabilidad. El algoritmo inicial fue propuesto por **Besl**, constaba de dos fases: identificar los puntos semilla, y aumentar la región basándonos en criterios preestablecidos como la proximidad de puntos o la planalidad de superficies. Este método es muy sensible al ruido y costoso en tiempo. Estos enfoques son altamente dependientes de los puntos semilla seleccionados. Una falta de precisión escogiendo estos puntos podría afectar a la segmentación y causar una infra-segmentación o una sobre-segmentación. Otra de las dificultades clave de este método es decidir si añadir puntos a una región dada.
- ***Unseeded región* (arriba-abajo).** Al comienzo, todos los puntos son agrupados en una sola región. El proceso de división empieza a dividir la región original en regiones más pequeñas. Mientras que una subregión supere cierto umbral preestablecido, la división continuará. Una limitación de este método es que podría haber sobre-segmentación y no trabaja bien al segmentar algunos objetos como árboles. La dificultad principal reside en decidir cuándo dividir. También, requieren mucho conocimiento a priori (modelos de objetos, número de regiones, etcétera), que normalmente es desconocido en escenas complejas.

Métodos basados en atributos

Se trata de enfoques robustos basados en agrupamiento de atributos en la nube de puntos. Se incluyen dos pasos distinguibles: el primero es la estimación de

los atributos. En el segundo paso, los puntos serán agrupados basándose en los atributos estimados.

Los métodos de agrupamiento ofrecen flexibilidad al adaptar la relación espacial y atributos para incorporar diferentes señales al proceso de segmentación. Una limitación de este proceso es que es altamente dependiente de la calidad de los atributos derivados. Los atributos de la nube de puntos deberán calcularse precisamente para producir la mejor separación entre las diferentes clases.

Los métodos basados en atributos son un enfoque robusto para agrupar puntos en regiones homogéneas. Sin embargo, estos métodos depositan confianza en la definición de vecindario entre puntos y la densidad de puntos de la nube. Otra limitación es el consumo en tiempo cuando se trabaja con atributos multidimensionales para una cantidad masiva de puntos.

Métodos basados en modelos

Utilizan formas geométricas primitivas (esfera, cono, plano, cilindro) para agrupar puntos. Los puntos que tienen la misma representación matemática (la misma primitiva asociada) son agrupados como un segmento.

Los métodos basados en modelos tienen principios matemáticos puros. Son rápidos y robustos a *outliers*. La limitación principal de estos métodos es su falta de precisión al tratar con diferentes fuentes de nubes de puntos.

Métodos basados en grafo

Estos métodos consideran la nube de puntos en términos de grafo. Un modelo simple es corresponder cada vértice con un punto de la nube, y las aristas conectan ciertos pares de puntos vecinos. Los métodos basados en grafo son precisos y han ganado popularidad para aplicaciones robóticas debido a su eficiencia. Un enfoque conocido es el algoritmo FH, es simple, eficiente y opera como el algoritmo de *Kruskal* para encontrar el **árbol generador minimal** (árbol de recubrimiento mínimo: un subgrafo del original que conecta todos los vértices con el menor número de aristas) en un grafo.

Golovinskiy utiliza kNN (*k Nearest Neighbors*, k vecinos más cercanos) para construir un grafo 3D en la nube de puntos. Este método introduce una función de penalización para incentivar una segmentación suave donde el primer plano está débilmente conectado con el fondo. Este método puede ser completamente automático, o interactivo con una interfaz de usuario, pero requiere conocimiento a priori sobre la localización de los objetos a ser segmentados.

Para comparar con otros métodos, los métodos basados en grafo pueden segmentar escenas complejas en nubes de puntos incluyendo ruido o densidad desigual de puntos con mejores resultados. Sin embargo, estos métodos normalmente no pueden ejecutarse en tiempo real. Algunos de ellos podrían necesitar entrenamiento o requerir sensores co-registrados y un sistema de cámara.

1.3.4.3 Estado del arte en técnicas de segmentación de nubes de puntos

Con el auge de la inteligencia artificial y, por consiguiente, del aprendizaje automático, han surgido nuevas y novedosas técnicas para segmentación que utilizan técnicas de aprendizaje profundo para realizar la tarea de la segmentación. A continuación, se exponen las principales y más conocidas.

PointNet

Desarrollado por investigadores de la Universidad de Stanford (Charles R Qi et al., 2016), este enfoque se posiciona como el estado del arte en todos los métodos de segmentación que utilizan aprendizaje profundo.

Esta propuesta pretende evitar la transformación previa de la nube en otro tipo de representación alternativo (véase la sección 1.3.3.2) para poder servir de entrada a la red neuronal.

Las nubes de puntos son estructuras simples y unificadas que evitan las irregularidades combinatorias (los puntos no están conectados de ninguna manera, por lo que no existen opciones para conectarlos, esto es, existe una invariabilidad a permutaciones) y la complejidad de las mallas, por tanto, es más fácil aprender patrones espaciales.

En las fases iniciales, cada punto es procesado independientemente. En la configuración básica, cada punto es representado por sus dimensiones (x, y, z). Dimensiones adicionales para cada punto (características) serán añadidas durante la propagación, enriqueciendo la tarea de segmentación/clasificación.

La clave de este enfoque reside en el uso de **una sola función simétrica**, *max-pooling*. La red aprende un conjunto de funciones de optimización/criterios que seleccionan puntos interesantes o informativos de la nube y codifican la razón de su selección. Las capas *fully-connected* del final agregan estos valores optimales aprendidos en un descriptor global para la forma completa (clasificación) o para predecir etiquetas a nivel de punto (segmentación). Lo más interesante es que la red aprende a resumir una nube de puntos de entrada en un conjunto escaso de puntos clave, que corresponde aproximadamente al esqueleto de los objetos según la visualización.

Dado que en una nube de puntos no existe un orden que pueda definirse entre los puntos, un aspecto a tratar es la invariabilidad a permutaciones, de ahí la función simétrica. Una función simétrica es invariante al orden de los valores de entrada. En PointNet, la red debe clasificar igualmente una forma constituida por puntos, independientemente del orden en que se inserten los puntos en la entrada de la misma.

Planteamiento del problema

Se trata del diseño de un marco de aprendizaje profundo que toma como entrada conjuntos de datos de puntos no ordenados: nubes de puntos en el sentido estricto.

Una nube de puntos es representada como un conjunto de puntos $\{P_i \mid i = 1, \dots, n\}$, donde cada punto P_i es un vector de coordenadas (x, y, z) además de información extra como el color, la normal, etcétera. Por simplicidad y claridad, solo se utilizan las coordenadas (x, y, z) como canales de información para el punto.

Las tareas que puede resolver la red son:

- **Clasificación.** La nube de entrada se toma de una forma u objeto segmentado de una escena de nube de puntos. Como salida se dan k valoraciones para las k posibles clases del problema.
- **Segmentación semántica.** La entrada puede ser un solo objeto (segmentación de partes) o un sub-volumen de una escena 3D para segmentación de objetos. La salida será una matriz de $N\ x\ M$ posiciones, donde N es el número de puntos de entrada y M las posibles categorías.

Arquitectura de PointNet

A continuación, se describen los tres principales bloques que componen PointNet (Charles R Qi et al., 2016).

- **Función de simetría para una entrada sin orden.** Para construir un modelo invariable a permutaciones de entrada, existen 3 estrategias: **1)** Ordenar la entrada en un orden canónico, esto es, todos los conjuntos tendrán definido el mismo operador de orden; **2**) Tratar la entrada como una secuencia para entrenar una **red neuronal recurrente** (Wu et al., 2018), siendo necesario aumentar los datos de entrenamiento con todo tipo de permutaciones (no es factible: N = 3.400.000 puntos, N! → infinito); **3)** Utilizar una función simétrica simple (*max-pooling*) para agregar la información de cada punto. En esta última, una función simétrica toma n puntos como entrada y da como salida un nuevo vector que es invariable al orden de entrada. Por ejemplo, los operadores + y * son funciones binarias simétricas.

Mientras que ordenar el conjunto es aparentemente una solución trivial, en espacios de alta dimensionalidad no existe un orden que sea estable con respecto a perturbaciones de puntos en el sentido general.

La idea es aproximar una función general definida en un conjunto de puntos aplicando una función simétrica en los elementos transformados en el conjunto:

$$f(\{x_1, \dots, x_n\}) \approx g(h(x_1), \dots, h(x_n)) \quad (1.10)$$

Empíricamente, el módulo básico es muy simple: se aproxima h con un perceptrón multicapa y g con una composición de una función de una sola variable y una función de *max-pooling*. Mediante una colección de funciones h, es posible aprender un número de funciones f para capturar diferentes propiedades del conjunto.

- **Agregación de información local y global.** La salida de la sección anterior forma un vector de características [f_1, …, f_k], que es una representación global del conjunto de entrada. Es posible entrenar fácilmente un perceptrón multicapa en las características de forma globales para realizar la clasificación. Sin embargo, la **segmentación** de puntos requiere una combinación de conocimiento local y global.

Tras calcular el vector de características globales de la nube, realimentamos las características por punto concatenando la característica global con cada una de las características por punto. Así, se extraen características que combinan la información local y global.

Con esta modificación, la red es capaz de predecir cantidades por punto que residen sobre la geometría local y la semántica global. Por ejemplo, los vectores normales para cada punto, validando que la red es capaz de resumir información del vecindario local al punto.

- **Red de conjunto de alineación.** El etiquetado semántico de la nube debe ser invariante a transformaciones geométricas (traslación, rotación, escalado). Por tanto, se espera que la representación aprendida en nuestro conjunto de puntos sea invariable.

Se predice una matriz de alineación con una mini-red (T-net) y se aplica esta transformación sobre las coordenadas de los puntos. La mini-red en sí misma se asemeja a la gran red (PointNet) y está compuesta por módulos básicos de extracción de características independientes del punto, *max-pooling* y *fully-connected*.

Esta idea puede extenderse a la alineación del espacio de características. Es posible insertar otra red de alineamiento de características de los puntos y predecir una matriz de transformación de características para alinear características de diferentes entradas de nubes de puntos.

La matriz de transformación en el espacio de características es de una dimensión mayor, lo que dificulta la optimización. Por tanto, se añade un término de regularización a la función de coste.

Se restringe la matriz de transformación para que sea semejante a la matriz ortogonal. En la imagen posterior se observa la arquitectura general de PointNet (véase Figura 1.33).

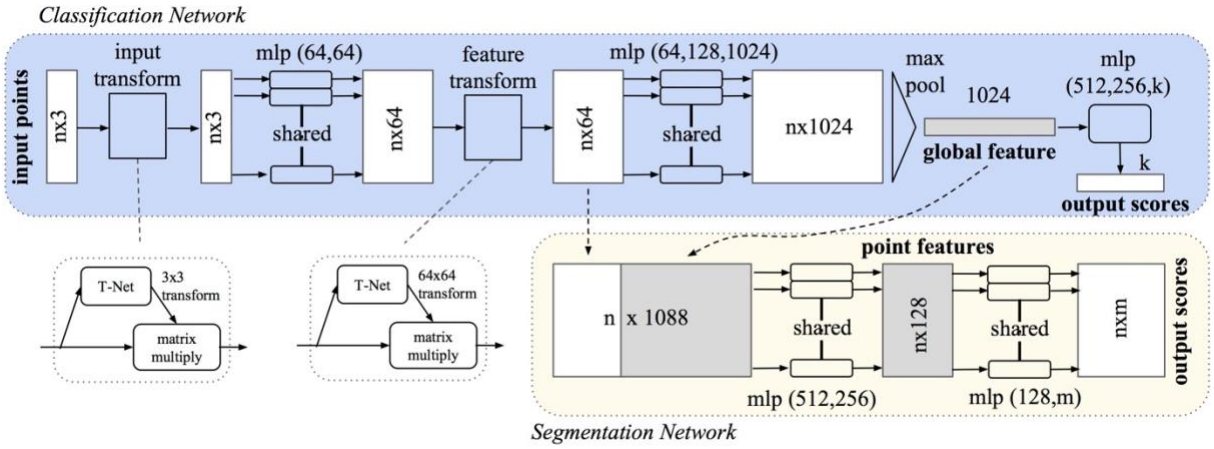

Figura 1.33: Esquema de PointNet. Fuente: Stanford

Detalles sobre la arquitectura

- **PointNet de clasificación.** La primera red de transformación es una mini PointNet que toma la nube de puntos en el formato de entrada y genera como salida una matriz 3x3, que será una matriz de transformación aplicable sobre todos los puntos originales. Está compuesta por un

perceptrón multicapa compartido (64, 128, 1024) para todos los puntos, un *max-pooling* sobre los puntos y dos capas *fully-connected* de tamaños de salida (512, 256) respectivamente. La matriz de salida es inicializada con la matriz identidad. Todas las capas, excepto la última, incluyen ReLU como función de activación y un operador de normalización de *batch*. La segunda red de transformación tiene la misma arquitectura que la primera excepto que la matriz de salida es una matriz de 64x64. Se usa *dropout* con una probabilidad de permanencia de 0.7 en la última capa *fully-connected*, es decir, la probabilidad de que se anule la salida de una neurona es de 0.3, para evitar el sobreajuste.

- **Red de segmentación de PointNet.** Se trata de una extensión a la red de clasificación. Las características locales de los puntos y la característica global extraída por la red de clasificación son concatenadas, resultando en M nuevos vectores de características para los M puntos de la nube a segmentar.

VoxNet

Se aborda el problema de predecir la etiqueta de clase de un objeto dado un *segmento* de una nube de puntos 3D (Maturana & Scherer, 2015), que podría tener ruido de fondo. Esta arquitectura aprende a extraer características y clasificar objetos representados como datos volumétricos.

Enfoque propuesto

Se recibe como entrada un segmento de nube de puntos. Normalmente se genera haciendo la intersección entre su caja envolvente y la nube de puntos 3D.

La tarea consiste en predecir la clase para el segmento.

- **Grid de ocupación volumétrico.** Existen 2 motivos principales para utilizar *grids* de ocupación: 1) permiten estimar eficientemente espacio libre, ocupado y desconocido de medidas de rango, incluso para medidas procedentes de distintos puntos de vista e instantes de tiempo. 2) Pueden ser manejados con estructuras de datos simples y eficientes.

- **Marco de referencia y resolución.** En la representación volumétrica, cada punto (x, y, z) es mapeado a coordenadas discretas de vóxel (i, j, k). El mapeado es un discretización uniforme, pero depende del origen, orientación y resolución del *grid* en el espacio. Para el **origen**, se asume que se da como entrada. Para la **orientación**, se asume que el eje Z del *grid* está alineado con la dirección de la gravedad (perpendicular a la superficie terrestre). Para la **resolución**, se adoptan dos estrategias diferentes según el dataset: 1) Para datos LiDAR, se usa una resolución espacial fijada de 0.1m para el vóxel en las tres direcciones del espacio. 2) Para otros datasets, la resolución es elegida de manera que el objeto de interés ocupe un subvolumen de 24^3 vóxeles.

VoxNet propone una **técnica de aumento de datos** consistente en aplicar entre 12 y 18 rotaciones alrededor del eje Z sobre la nube de puntos de entrada, con el objeto que incrementar el tamaño del dataset, y que la nube aprenda a clasificar un objeto desde sus diferentes puntos de vista.

Reflexión

Este enfoque pretende la **clasificación** de un *segmento* de la nube, es decir, el paso inmediatamente posterior a la **segmentación**. El hecho de esperar un objeto ya segmentado, esto es, un subconjunto de puntos de la nube original, como entrada supone que esta propuesta no sirva para desarrollar el proyecto, aunque sí se tomarán algunas ideas de utilidad del mismo.

PointNet++. Aprendizaje profundo de características jerárquicas en conjuntos de puntos del espacio métrico

Por su diseño, **PointNet** no captura estructuras locales inducidas por el espacio métrico donde residen los puntos, **limitando** su habilidad para reconocer patrones detallados y generalidad a escenas complejas.

Este enfoque introduce una red neuronal jerárquica que aplica PointNet recursivamente en una partición anidada del conjunto de puntos de entrada (Charles Ruizhongtai Qi et al., 2017).

Introducción

Un tipo particularmente importante de conjunto de puntos geométricos es el capturado por escáneres 3D. Como un conjunto, debe ser invariable a las permutaciones de sus miembros.

Explotar estructuras locales ha demostrado ser importante para el éxito de las arquitecturas convolucionales. Una red convolucional toma datos definidos en *grids* regulares como entrada y es capaz de capturar progresivamente características a escalas cada vez mayores a lo largo de una jerarquía de resolución múltiple.

La idea general de PointNet++ es simple: primero se particiona el conjunto de puntos en regiones locales que se solapan por la distancia métrica del espacio subyacente. De manera similar a las redes convolucionales, se extraen características locales capturando estructuras geométricas de pequeños vecindarios.

El diseño de PointNet++ tiene que solucionar dos problemas: 1) cómo particionar el conjunto de puntos, y 2) cómo abstraer conjuntos de puntos o características locales mediante el aprendizaje de características locales.

Como bloque básico, PointNet abstrae conjuntos de puntos locales o características en representaciones de más alto nivel. Desde este punto de vista, PointNet++ aplica PointNet recursivamente en un particionamiento anidado del conjunto de puntos de entrada.

Algo pendiente es cómo solucionar el particionamiento solapante del conjunto de puntos. Cada partición está definida como una esfera de vecindario en el espacio euclídeo subyacente, cuyos parámetros incluyen la localización del centroide y escala. Para cubrir el conjunto completo, los centroides son seleccionados entre el conjunto de entrada por un algoritmo de punto lejano (FPS, *Furthest Point Sampling*).

PointNet toma ventaja de vecindarios a diferentes escalas (múltiples) para conseguir a la vez robustez y captura de detalles. Ayudado con *dropout* aleatorio de la entrada durante el entrenamiento, la red aprende a pesar de forma adaptativa

patrones detectados en diferentes escalas y a combinar características multi-escala de acuerdo a los datos de entrada.

Estado del problema

Se supone que $X = (M, d)$ es un espacio métrico discreto cuya métrica es heredada de un espacio euclídeo $\mathbb{R}^n$, donde $\mathrm{M} \subseteq \mathbb{R}^n$ es el conjunto de puntos y d es la distancia métrica. Además, la densidad de M en el espacio euclídeo será desigual. El interés reside en aprender funciones de conjunto, f, que toman X como entrada (con características adicionales por punto) y produce información de interés para X. En la práctica, f puede ser una función de clasificación que asigna una etiqueta a X o una función de segmentación que asigna una etiqueta por punto a cada miembro de M.

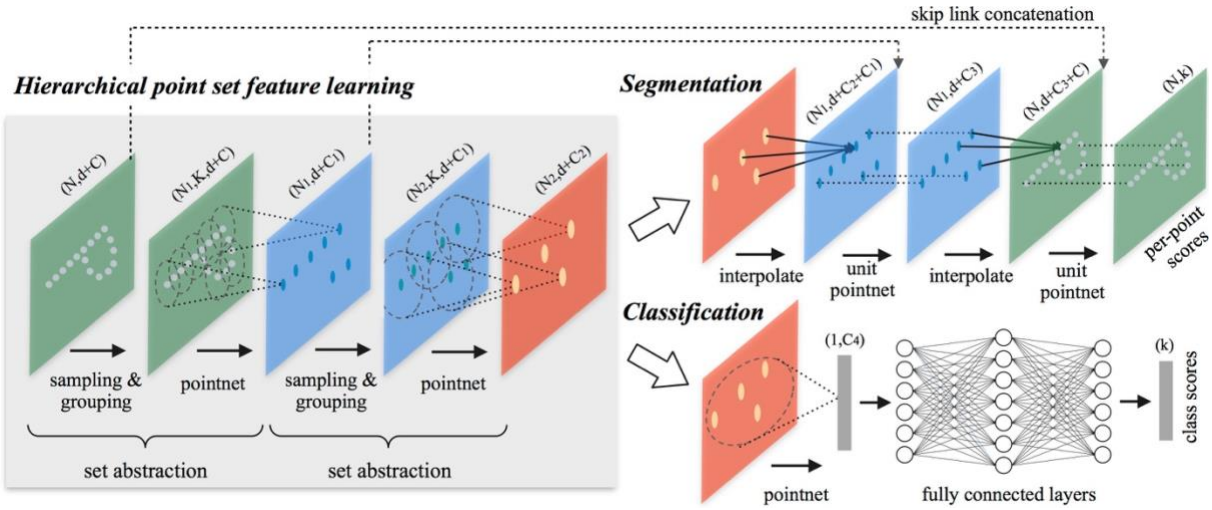

Figura 1.34: Esquema de PointNet++. Fuente: Stanford

Método

PointNet++ puede verse como una extensión de PointNet con una estructura jerárquica añadida, tal y como se observa en la Figura 1.34). PointNet, en sí, carece de capacidad para captar el contexto local a diferentes escalas. Se presenta un marco de aprendizaje de características jerárquicas para resolver esta limitación.

- **Aprendizaje de características de conjuntos de puntos jerárquicos.** Mientras PointNet utiliza una única operación *max-pooling* para agregar el

conjunto de puntos completo, la nueva arquitectura construye una agrupación jerárquica de puntos y progresivamente abstrae regiones locales más y más grandes en la jerarquía. Esta estructura jerárquica está compuesta por un número de niveles de abstracción de conjunto. En cada nivel, un conjunto de puntos es procesado y abstraído para producir un nuevo conjunto con menos elementos. El nivel de abstracción está formado por 3 capas clave.

- **La capa de muestreo (*sampling*).** Selecciona un subconjunto de los puntos de entrada, que definen los centroides de las regiones locales.
- **La capa de agrupamiento (*grouping*).** Construye conjuntos de región local encontrando puntos 'vecinos' alrededor de los centroides.
- **La capa PointNet.** Se utiliza una red PointNet básica para codificar patrones de región local en vectores de características.

Un nivel de abstracción de conjunto toma una matriz $N\ x\ (d\ +\ C)$ como entrada, de N puntos, d-dimensionales en el espacio métrico y C-dimensionales en cuanto a características. Genera como salida una matriz de $N'\ x\ (d\ +\ C')$ de N' puntos submuestreados con d-dimensional coordenadas y nuevos vectores C'-dimensionales de características, resumiendo el contexto local.

- **Aprendizaje de características robusto bajo densidad no-uniforme de muestreo.** Es común que un conjunto de puntos tenga una densidad no uniforme en diferentes áreas. Esto representa un reto para el aprendizaje de características. Las características aprendidas en datos de nubes de puntos de **alta densidad** podrían no generalizar bien cuando los datos de entrada son **dispersos**, y viceversa. Idealmente, se trata de explorar tan cerca como sea posible en un conjunto de puntos para capturar los detalles más finos en regiones muestreadas densas. Sin embargo, esto está prohibido en regiones de baja densidad. En este caso, se deben buscar patrones de mayor escala en un vecindario más grande. Para

conseguir esto, se proponen unas capas de PointNet adaptativas a la densidad que aprenden a combinar características de regiones a diferentes escalas cuando la densidad de muestreo de entrada cambia.

En la sección anterior, cada nivel de abstracción contenía agrupamiento y extracción de características para una sola escala. En PointNet++, cada nivel de abstracción extrae múltiples escalas de patrones locales y los combina de manera inteligente de acuerdo a las densidades de puntos locales. En términos de agrupamiento de regiones locales y combinación de características a diferentes escalas, se proponen dos tipos de capas de densidad adaptativa.

- **Agrupamiento multi-escala.** Se aplican capas de agrupamiento con diferentes escalas seguida por redes PointNet para extraer características a cada escala. Las características a diferentes escalas se concatenan para formar una característica multi-escala. La red se entrena para aprender una estrategia optimizada de combinación de características. Esto se hace mediante *dropout* de puntos de entrada con una probabilidad aleatoria para cada instancia. En el test, se usan todos los puntos.
- **Agrupamiento multi-resolución.** El enfoque anterior es costoso computacionalmente, ya que ejecuta PointNet en vecindarios a gran escala para cada punto centroide. Cuando la densidad de una región local sea baja, el primer vector será menos fiable que el segundo vector, ya que la subregión que calcula el primer vector contiene puntos más dispersos y sufre más deficiencia de muestreo. En ese caso, el segundo vector tendrá que ponderarse más. Si por el contrario la densidad es mayor, el primer vector tendrá información sobre detalles más precisos a resoluciones más altas recursivamente a niveles más bajos.

Etiquetado de nubes de puntos con redes convolucionales 3D

Este enfoque parte de una voxelización de la nube de puntos y aplicación de redes de convolución 3D (Huang & You, 2016). No se requiere información adicional en cada punto como las normales, el color, etcétera.

Dada la naturaleza irregular de una nube de puntos, se propone la voxelización para obtener una representación regular de la nube. Si bien se conoce que este método genera imprecisión.

La arquitectura de red propuesta es la siguiente (véase Figura 1.35).

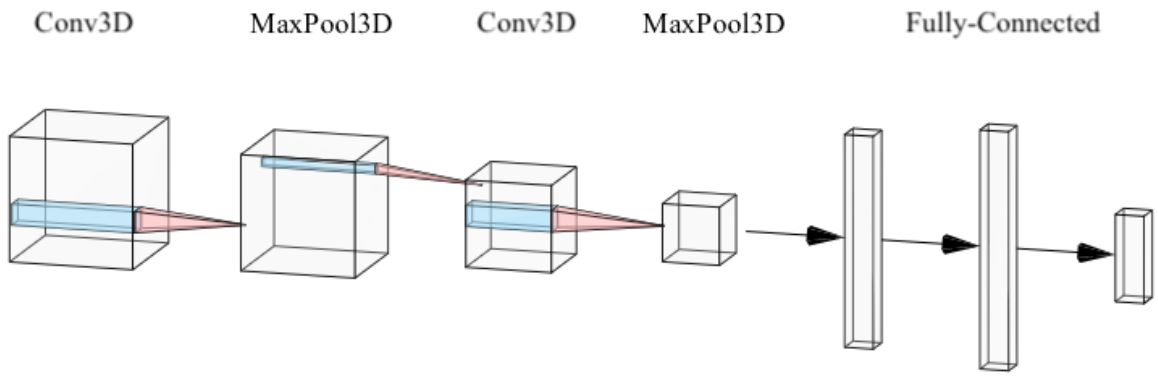

Figura 1.35: Red Neuronal Convolucional 3D

En lugar de voxelizar la nube completa, se generan una serie de *grids* se vóxeles a partir de los puntos clave escogidos; *grids* con una dimensión fija de $N \; x \; N \; x \; N$ que podrán servir como entrada a la red neuronal. Una vez hecha la propagación, la red emite una etiqueta para el *grid* de vóxeles completo, que será asignada únicamente al punto central del vóxel: el punto a partir del cual se generó el *grid* completo.

Esta última alternativa será la escogida para realizar una experimentación y probar la segmentación de nubes de puntos.

1.4 Requisitos iniciales

El principal objetivo de este proyecto es estudiar la segmentación de objetos en nubes de puntos desde un enfoque de aprendizaje profundo. Además, se exponen las distintas técnicas clásicas existentes hoy en día para realizar dicha tarea, así como algunas que están en el estado del arte, y que utilizan aprendizaje profundo.

Al hibridar **técnicas de inteligencia artificial con informática gráfica**, uno de los objetivos es servir de referencia a cualquier principiante en aprendizaje profundo con interés en el área que tenga el propósito de entrar en materia de una manera progresiva con un nivel de detalle ascendente. Aunque el objetivo es la segmentación, la explicación de conceptos previos es fundamental y es el resultado de un proceso de estudio de la materia de varios meses, unos contenidos que no se imparten durante el grado. Por otra parte, la formación en el tratamiento inteligente de información es algo que a grandes rasgos se desarrolla durante el grado, por tanto, otro aspecto es dar a descubrir al lector la existencia de algunas áreas de trabajo muy interesantes y que a priori son desconocidas.

Se propone el desarrollo de un sistema de etiquetado de nubes de puntos que toma como base el aprendizaje profundo.

El sistema *ha de* recibir una nube de puntos como entrada y para cada punto de esa nube *debe* generar una etiqueta, indicando a qué **clase semántica** pertenece. Se *debe* diseñar una red neuronal que *sea capaz* de aceptar como entrada la nube, para lo que habrá que resolver la irregularidad inherente a las nubes de puntos. Para tal fin, se *debería* diseñar una estrategia que permita generar un conjunto de entrenamiento, y entrenar la red neuronal para **extraer patrones en la nube**; patrones con los que más tarde *debería* ser capaz de segmentar una nube de puntos completa.

La experimentación es una tarea costosa y consume una gran cantidad de recursos en tiempo y espacio, por lo que el sistema *deberá* hacer un uso eficiente de los recursos hardware y trabajar con el mejor rendimiento posible. Al tratarse de una

tarea experimental, se *deberá* estudiar y proponer una alternativa que sea un compromiso entre la maximización de la calidad algorítmica y la minimización de errores durante el proceso.

1.5 Alcance

El presente trabajo plantea una experimentación con una arquitectura de red neuronal convolucional que tiene como objetivo resolver la segmentación semántica en nubes de puntos tridimensionales. Para lo cual, se establecen los siguientes contenidos entregables:

- **Introducción al aprendizaje automático y aprendizaje profundo.** Se incluye una explicación detallada de todos los aspectos y conceptos propios del aprendizaje profundo, desde los orígenes hasta las arquitecturas de red neuronal presentes hoy en día, y utilizadas en numerosas aplicaciones.
- **Introducción al tratamiento de la información espacial.** Se realiza un repaso de los principales formatos de representación de la información espacial, tanto 2D como 3D, llevando a cabo una revisión detallada sobre **nubes de puntos**; así como un análisis de los dispositivos de captación/generación de información espacial.
- **Análisis del estado del arte de las principales técnicas de segmentación en nubes de puntos tridimensionales.** Análisis y explicación de las principales técnicas para segmentación en nubes de puntos, así como un breve análisis de las carencias y ventajas de cada uno de ellos.
- **Material utilizado.** Descripción de los materiales necesarios tanto *hardware* como *software*, así como el *dataset* utilizado en la experimentación.
- **Descripción de la técnica utilizada.** De entre todas las técnicas estudiadas en secciones previas, se selecciona una de ellas para desarrollar una experimentación y concluir al respecto. Así, se incluye una explicación de las estrategias necesarias como el preprocesamiento de datos, la voxelización, y la propuesta de arquitectura de red neuronal.

- **Experimentación con la técnica utilizada.** Se realizan un total de 9 experimentos usando el *dataset Semantic 3D* y se concluye al respecto, indicando los aspectos positivos y negativos de la técnica, así como un análisis de resultados de cada experimento.
- **Dataset utilizado para la experimentación.** Para desarrollar la técnica se ha hecho uso de un dataset de pruebas destinado a segmentación semántica de nubes de puntos tridimensionales: *Semantic 3D* (http://www.semantic3d.net/). Se trata de un *dataset* de 15 nubes de puntos de entrenamiento con alrededor de 4.000 millones de puntos etiquetados en 8 categorías semánticas. Está disponible para su descarga en formato ASCII.
- **Código fuente desarrollado para realizar los experimentos.** Se incluye junto a la entrega el código fuente desarrollado para el trabajo fin de grado. En la sección 5.1 se detalla el contenido.

1.6 Hipótesis y restricciones

El TFT se define como una asignatura de 12 créditos, lo que supone que la duración total del proyecto será de 300 horas, incluyendo todas las etapas del ciclo de vida, con la excepción del mantenimiento. Por consiguiente, la principal restricción aplicable es la limitación de la duración del trabajo.

Para la realización del trabajo se ha hecho uso y disfrute de una beca ministerial de colaboración con el Departamento de Informática, con acceso a cualquier recurso hardware o software necesario, por lo que no existen restricciones de coste. Por otra parte, las restricciones de calidad están limitadas a conseguir unos resultados similares a las principales técnicas en el estado del arte.

1.7 Estudio de alternativas y viabilidad

En esta sección se realiza un análisis las diferentes alternativas que están disponibles para segmentación de nubes de puntos utilizando técnicas de aprendizaje profundo, que ya han sido descritas de una forma detallada en la

sección de estado del arte, con el objeto de establecer qué criterios han llevado a tomar la decisión de decantarse por una técnica en concreto.

1.7.1 Técnicas clásicas

Las técnicas clásicas, como las basadas en borde, de crecimiento de región, etcétera, han sido ampliamente estudiadas y utilizadas durante décadas de investigación, y finalmente la conclusión que se extrae es que **son extremadamente imprecisas** y también son bastante **costosas computacionalmente**, resultando en una segmentación muy ineficaz e ineficiente. Por tanto, se decide **descartar** esta opción.

El propósito es el estudio de un **marco de segmentación que se asiente sobre el aprendizaje profundo**. De tal modo, la tarea de segmentación debe realizarse mediante la aplicación de un algoritmo basado en redes neuronales, extrayendo sus propias características con las que mejor segmente, evitando la especificación formal por parte del experto; esto es, su propia representación del conocimiento, y no sesgando el sistema desde un principio como ocurre con el aprendizaje automático tradicional o las técnicas clásicas, en las que la segmentación está limitada a información proporcionada por el usuario como la localización de los puntos de entrada, la estimación de las normales, el ensamblado de primitivas, etcétera.

La genialidad del aprendizaje profundo reside en la construcción automática de representaciones del conocimiento abstraídas de la realidad que permitan resolver la tarea de la manera más acertada. La tendencia es a emplear algoritmos de aprendizaje automático, sirviéndose estos de características de entrada proporcionadas por los humanos: características que los humanos son capaces de comprender. En cambio, la naturaleza jerárquica del conocimiento extraído permite establecer relaciones entre conceptos, obteniéndose cada vez conceptos más complejos, atacando la problemática desde una perspectiva difícilmente o imposible de visualizar para un humano experto.

1.7.2 Estado del arte en técnicas de segmentación

Las principales técnicas que están en el estado del arte utilizan aprendizaje profundo para realizar la segmentación; la principal diferencia reside en que usan diferentes representaciones para los datos de entrada y también están diseñadas para una arquitectura y aplicación concretas.

- **Etiquetado de nubes de puntos con redes neuronales convolucionales 3D.** (Huang & You, 2016) Esta alternativa parte de realizar voxelizaciones en entornos locales de la nube, y realizar con ello la segmentación sirviendo a la red neuronal como entrada estos *grids* de vóxeles generados. Es necesario escoger una *precisión* para los vóxeles, de manera que conserven el máximo nivel de detalle posible tras la voxelización. Esta técnica minimiza el conocimiento previo necesario a conocer únicamente la etiqueta semántica de cada punto; no está diseñada especialmente para trabajar con nubes de exteriores/interiores o formas **CAD**, dejando libertad a su usuario para diseñar la aplicación al gusto. Esta será la alternativa que defina la línea de trabajo principal de este proyecto.
- **PointNet (PointNet++).** Tanto la primera como la segunda versión de PointNet (Charles R Qi et al., 2016; Charles Ruizhongtai Qi et al., 2017) están dedicadas tanto a clasificación como a segmentación semántica. El problema de esta arquitectura es que está diseñada y optimizada para el trabajo con nubes muestreadas de formas **CAD** o captadas mediante sensores en interiores, como por ejemplo un sensor **Kinect**. Lo que hacen es realizar un muestreo sobre la nube original, quedándose con los puntos más significativos de dicha nube (esqueletonización), y servir como entrada a la red estos puntos muestreados para que consiga extraer patrones espaciales con los que clasificar/segmentar. Para exteriores, como es el caso, este enfoque no es suficiente. Una nube de puntos generada mediante LiDAR aéreo puede contener varios (cientos) millones de puntos, y la distancia entre ellos suele ser de unos 30 centímetros. Un muestreo insuficiente (entendiendo como insuficiente que escoge puntos significativos, pero no escoge todos los que lo son) por parte de este

sistema podría hacer que se perdiesen puntos de vital importancia para segmentar correctamente 'coches', por ejemplo.

- **VoxNet.** Esta arquitectura, aunque sencilla, no sirve para resolver la segmentación en absoluto, pues está dedicada a clasificación de objetos que parten de modelos de nubes de puntos (Maturana & Scherer, 2015). Requiere segmentos de la nube, por lo tanto, se parte de que previamente ya se ha realizado una segmentación y el siguiente paso es clasificar estos segmentos extraídos. En cambio, se proponen una serie de medidas para tratamiento del *dataset* que son de gran interés para incorporar a nuestra propuesta.

1.8 Descripción de la solución propuesta

La presente propuesta parte del **etiquetado de nubes de puntos utilizando redes neuronales convolucionales con capas de convolución tridimensionales** (Huang & You, 2016)**.** Se trata de la alternativa más viable en cuanto al trabajo de nubes procedentes de escaneados en exteriores (LiDAR, etcétera), algo que no ha sido resuelto aún resuelto por ninguna arquitectura.

Por otra parte, se trata de la alternativa que requiere menos conocimiento a priori por parte de un experto, pues la única información necesaria para el entrenamiento es la nube de puntos original, con coordenadas por punto (x, y, z) y la etiqueta asociada a ese punto de entre las diferentes contempladas en el problema en concreto, dejándose a un lado información más compleja como la normal en cada punto, etcétera. Durante el **test**, el sistema acepta una nube de puntos no etiquetada como entrada, y genera como salida una etiqueta para cada uno de los puntos presentes en la nube.

Esta arquitectura parte de realizar **voxelizaciones locales**, generando *grids* de vóxeles de tamaño fijo en entornos locales de la nube tal resultando en volúmenes como en la Figura 1.36. El objetivo es que cada punto pueda etiquetarse utilizando información que es capaz de inferirse a partir de los vecinos. De la nube original se extrae un subconjunto de puntos, y para cada uno de ellos se genera un *grid* de vóxeles de ocupación tomando para ello información de los vecinos.

Se define un tamaño de vóxel, t, y un número de vóxeles $N \times N \times N$ en cada *grid*. Así, como cada *grid* se generará a partir de un punto con coordenadas (x, y, z), cada grid encerrará una región del espacio de iguales dimensiones al resto teniendo como centro el punto seleccionado. Estas regiones serán las que se voxelizarán posteriormente, resultando en espacios discretos en el que cada subregión discreta encierra un conjunto arbitrario de puntos y es resumida o representada como una unidad volumétrica.

Así, esta propuesta permite segmentar una nube de puntos completa con el mínimo conocimiento (únicamente se necesitan las coordenadas espaciales de cada punto) y de una manera muy sencilla e intuitiva, si bien para el entrenamiento se necesitan nubes presegmentadas (ya etiquetadas) pues se trata de un tipo de aprendizaje supervisado.

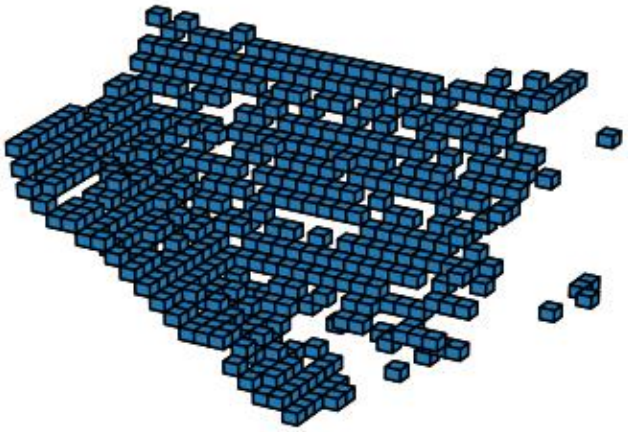

Figura 1.36: Voxelización de un entorno local

1.9 Material y métodos

Al tratarse de un trabajo **teórico-experimental**, se necesitan unos recursos que permitan estudiar la problemática y emitir un juicio de evaluación. En este caso, el recurso principal de partida es el *dataset* de nubes de puntos etiquetadas semánticamente que permitan realizar entrenamiento y posterior segmentación con un enfoque de redes neuronales.

Como es conocido, estos algoritmos requieren de una cantidad muy extensa de datos para entrenarse correctamente. Por tanto, cuanto más voluminoso el *dataset* de partida, mayor es el interés para el sistema pues siempre se podrá muestrear y reducir su tamaño en caso de que fuese necesario. En cambio, partir de un conjunto de datos pequeño requiere interpolar sobre los datos existentes para aumentar el tamaño de este, lo cual carece de trivialidad.

Para este proyecto se ha utilizado el *dataset* **Semantic 3D** (http://www.semantic3d.net/)**.**

1.9.1 Dataset. Semantic 3D

Se trata de un *framework* para la evaluación de algoritmos de segmentación/clasificación de nubes de puntos que contiene en torno a 4.000 millones de puntos con dimensiones no normalizadas etiquetados manualmente. El *dataset* de entrenamiento está compuesto por un total de 15 ficheros de nubes de puntos de escenas rurales, urbanas y suburbanas como la que se visualiza en la Figura 1.37, etiquetadas convenientemente para el entrenamiento con 8 categorías semánticas:

1. Terreno artificial.
2. Terreno natural.
3. Vegetación alta.
4. Vegetación baja.
5. Edificios.
6. Paisaje accidentado.
7. Artefactos de escaneo.
8. Coches.

Los 15 ficheros de nubes de puntos que componen el conjunto de entrenamiento están almacenados en formato de texto plano *.txt*, y son los siguientes.

1. untermaederbrunnen_station3_xyz_intensity_rgb.txt
2. bildstein_station3_xyz_intensity_rgb.txt
3. bildstein_station5_xyz_intensity_rgb.txt
4. domfountain_station1_xyz_intensity_rgb.txt
5. domfountain_station2_xyz_intensity_rgb.txt
6. domfountain_station3_xyz_intensity_rgb.txt
7. neugasse_station1_xyz_intensity_rgb.txt
8. sg27_station1_intensity_rgb.txt
9. sg27_station2_intensity_rgb.txt
10. sg27_station4_intensity_rgb.txt
11. sg27_station5_intensity_rgb.txt
12. sg27_station9_intensity_rgb.txt
13. sg28_station4_intensity_rgb.txt
14. untermaederbrunnen_station1_xyz_intensity_rgb.txt
15. bildstein_station1_xyz_intensity_rgb.txt

Cada fichero contiene un número arbitrario de puntos, uno por línea, con la siguiente estructura:

# punto	X	Y	Z	Intensidad	R	G	B
1	20.36	40.376	-2.402	-245	23	34	76
…	…	…	…	…	…	…	…
N	x_N	y_N	z_N	i_N	r_N	g_N	b_N

Tabla 1.2: Estructura de un fichero de nubes de puntos ASCII en formato .txt

Las **coordenadas** de cada punto (x, y, z) estan definidas sobre un espacio métrico $X = (M, d)$, donde M representa al conjunto de puntos y d una distancia métrica definida sobre los puntos. El origen de coordenadas $(0, 0, 0)$ de la escena se sitúa en la posición del escáner LiDAR (TLS) en el espacio escaneado.

La componente **intensidad** es un canal de información capturado por el sensor LiDAR que proporciona una medida de la **reflectividad** de la superficie de los objetos capturados, mientras que las componentes **r**, **g**, **b** corresponden respectivamente al nivel en el intervalo $[0, 255]$ de **rojo**, **verde** y **azul** respectivamente de la superficie del objeto escaneado.

Para cada uno de los ficheros enumerados, existe su correspondiente fichero de **etiquetas**. En estos ficheros se tiene en cada línea la etiqueta semántica de cada punto. Por ejemplo, para el fichero *bildstein_station3_xyz_intensity_rgb.txt*, la etiqueta del punto en la línea 57 aparece en la línea 57 del fichero *bildstein_station3_xyz_intensity_rgb_labels.txt*. En este segundo fichero únicamente aparecen etiquetas, esto es, un número entero en el rango $[1, 8]$ por línea.

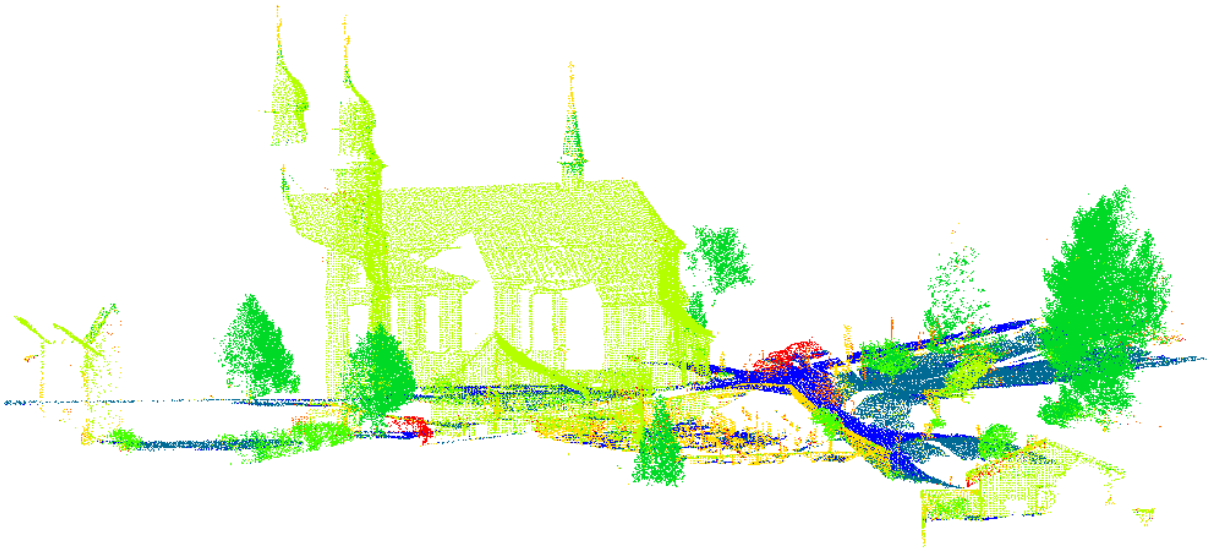

Figura 1.37: Nube de puntos de Semantic 3D

Es importante señalar que tal y como está compuesto este *dataset*, no puede servir como entrada al sistema directamente debido a la alta densidad de puntos, por lo que hay que realizar un **preprocesamiento** sobre el mismo.

1.9.2 Hardware

En cuanto al *hardware* empleado, se necesita un procesador especializado en cálculos vectoriales y matriciales, como el procesador gráfico. Una red neuronal puede representarse vectorialmente, y tanto la propagación hacia adelante como el algoritmo de entrenamiento (*back-propagation*) requieren de operaciones matriciales como la multiplicación de matrices. Con una arquitectura dedicada a cálculo vectorial, es posible optimizar esta operación.

El único dispositivo de procesamiento vectorial dentro de un computador es el procesador gráfico, con una arquitectura SIMD (*Simple Instruction Multiple Data*). Normalmente, este procesador suele constar de varios cientos (miles) de núcleos, donde cada uno de ellos realiza la misma operación que el resto de núcleos sobre datos diferentes. Es por ello que en su mayoría están dedicados a informática gráfica, ya que pueden procesar un conjunto muy grande de vértices en un lapso de tiempo.

Con la evolución del hardware gráfico, los programadores advirtieron la posibilidad de realizar programación de propósito general usando la **GPU (*Graphics Processor Unit*)**, esto es, procesamiento masivo de datos que no implicaba *rendering*, obteniendo como resultado un paralelismo real en la ejecución. Al principio, esta tarea se hacía escribiendo *shaders* directamente usando una sintaxis lejos de lo trivial, siendo necesario realizar llamadas a una API dedicada a gráficos como OpenGL (*Open Graphics Library*). Con el paso de los años, las compañías de fabricación fueron cambiando su asentamiento desde la dedicación exclusiva a *rendering* e informática gráfica en general, a otras áreas como la inteligencia artificial y el aprendizaje profundo, abriendo nuevas puertas a los desarrolladores y proponiendo *frameworks* como CUDA (*Compute Unified Device Architecture*, NVIDIA) o OpenCL (*Open Computing Library*, Apple) para la programación de propósito general usando la GPU. Con este avance, se dio paso a muchas tareas todavía imposibles como el entrenamiento de redes neuronales, por la cantidad de operaciones entre matrices inherente a este paradigma de procesamiento.

Así, otro pilar muy importante en cuanto a material será un hardware potente para realizar el desarrollo del software (CPU, memoria, etcétera), como también una GPU NVIDIA compatible con la tecnología CUDA con la que poder entrenar la red.

Para desarrollar la **experimentación**, se hará uso de una máquina con un sistema operativo Ubuntu Server 18.04, con un procesador Intel Core i7 8700K, 64GB de memoria RAM, y una tarjeta gráfica NVIDIA Titan Xp para entrenar los distintos modelos de red neuronal.

1.10 Tecnologías utilizadas

1.10.1 Sistema operativo y lenguaje de programación

El desarrollo de software se va a realizar sobre un ordenador con un sistema operativo Ubuntu Linux 18.04 (https://www.ubuntu.com/, Figura 1.39) ubicado en un laboratorio de investigación en la universidad. El lenguaje de programación escogido es Python (https://www.python.org/, Figura 1.38), en su versión 3.7, instalado mediante Anaconda 3 (https://www.anaconda.com/, Figura 1.40) en el sistema operativo.

Figura 1.38: Logotipo del lenguaje de programación Python. Fuente: Google

Figura 1.39: Logotipo de Ubuntu Linux. Fuente: Google

Figura 1.40: Logotipo de Anaconda 3. Fuente: Google

Para trabajar en el desarrollo desde casa, se hará uso de un ordenador Apple iMac de finales de 2014 con un sistema operativo macOS Mojave (véase Figura 1.41), y haciendo uso del código compartido con la otra máquina y que está alojado en un repositorio remoto.

Figura 1.41: Logotipo de macOS Mojave. Fuente: Google

1.10.2 Elección del framework de aprendizaje profundo

Al tratarse de redes neuronales, es necesario escoger un *framework* para aprendizaje profundo de entre todos los que están disponibles para el lenguaje Python (Tensorflow, PyTorch, Theano, Keras, Caffe2, etcétera). De lo contrario, sería necesario implementar la red a bajo nivel utilizando alguna librería de cálculo vectorial/matricial como NumPy.

Con el objeto de evitar tediosos cálculos e implementaciones a bajo nivel, surgen los *frameworks* de aprendizaje profundo que permiten desarrollar algoritmos basados en redes neuronales de una manera declarativa, esto es, no es necesario definir cómo se realizan las diferentes operaciones (multiplicaciones, cálculo de derivadas parciales, etcétera) a bajo nivel; únicamente se realizan llamadas a funciones y métodos de alto nivel que cumplen con el objetivo requerido.

En este caso, por su sintaxis sencilla y su abundante documentación, y por ser uno de los *frameworks* más utilizados a nivel mundial (lo que permitirá tener *feedback* de otros usuarios en caso necesario) para aprendizaje profundo, se escoge **PyTorch** (https://pytorch.org, Figura 1.42), desarrollado por Facebook AI (*Facebook Artificial Intelligence*).

Figura 1.42: Logotipo de PyTorch. Fuente: Google

PyTorch está dotado para hacer uso de la tecnología **CUDA** (https://www.nvidia.es/object/cuda-parallel-computing-es.html, Figura 1.43) sobre un procesador gráfico fabricado por **NVIDIA** (https://www.nvidia.com/es-es/). La tecnología CUDA es un paradigma de programación que permite paralelismo real en tareas de programación de propósito general usando el procesador gráfico (GPGPU, *General Purpose Computing on GPUs*) de una manera sencilla y eficiente sin necesidad de recurrir a métodos clásicos como la utilización del *fragment shader*. De

esta manera, tenemos acceso a computación HPC (*High Performance Computing*) de una manera relativamente económica y sin necesidad de ensamblar potentes *clústeres* con un hardware excesivamente caro.

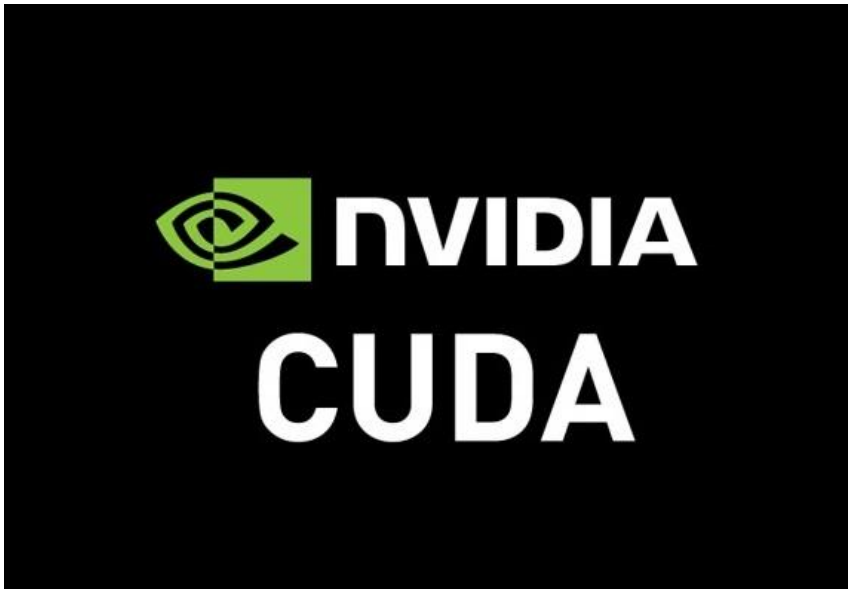

Figura 1.43: Tecnología NVIDIA CUDA. Fuente: Google

1.10.3 Entorno de desarrollo

El desarrollo y depuración de software se ha realizado empleando para ello el IDE (*Integrated Development Environment*) PyCharm de la compañía Jetbrains (https://www.jetbrains.com/pycharm/, Figura 1.44), dedicado exclusivamente a desarrollo de código en el lenguaje de programación Python.

Figura 1.44: Logotipo del IDE PyCharm. Fuente: Google

1.10.4 Gestión del código

El código se va a mantener mediante un sistema de control de versiones (CVS, *Control Version System*), *Git* (Figura 1.45), universalmente conocido y utilizado por la mayoría de los desarrolladores a nivel mundial, a la vez que se va a hacer uso de los repositorios de la compañía GitHub (https://github.com/, Figura 1.46), recientemente adquirida por Microsoft, para depositar el código y poder trabajar en paralelo desde diferentes máquinas.

Figura 1.45: Logotipo de git. Fuente: Google

Figura 1.46: Logotipo de GitHub

1.11 Metodología de desarrollo de software

1.11.1 Desarrollo incremental

Como metodología de desarrollo de software se ha optado por el **desarrollo incremental**. Se trata de un proceso de desarrollo de software que se creó como solución a las debilidades que presentaba el modelo de desarrollo tradicional en cascada (S. Pressman, 2006).

El proyecto se planifica en distintos bloques temporales que se denominan iteraciones. En cada iteración se repite un determinado proceso que aporta un

resultado más completo sobre el producto final, de manera que el destinatario del producto irá recibiendo beneficios del producto de forma creciente. Para lograr esto, cada requerimiento debe tener un completo desarrollo en una única iteración que debe incluir pruebas y una documentación para que el equipo pueda cumplir con todos los objetivos que sean necesarios y esté listo para ser dado al cliente. Así se evita tener arriesgadas actividades en el proyecto finalizado.

Lo que se busca es que en cada iteración los componentes logren evolucionar el producto dependiendo de los completados en iteraciones anteriores, agregando más opciones de requisitos y logrando así una mejora más completa.

1.11.2 Ciclo de vida

La idea principal detrás de la mejora iterativa es desarrollar un proyecto de forma incremental, permitiéndole al desarrollador aventajarse de lo que se ha aprendido a lo largo del desarrollo anterior, incrementando versiones entregables del sistema.

Los pasos claves en el proceso son comenzar con una implementación simple de los requerimientos del sistema, e iterativamente mejorar la secuencia evolutiva de versiones hasta que el sistema completo esté implementado. En cada iteración, se realizan cambios en el diseño y se agregan nuevas funcionalidades y capacidades al sistema.

El modelo parte básicamente de dos premisas: 1) el usuario nunca sabe exactamente qué necesita para satisfacer sus necesidades, y 2) durante el desarrollo, los procesos tienden a mutar.

- **Etapa de inicialización.** Se crea una versión del sistema. La meta de esta etapa es crear un producto con el que el usuario pueda interactuar, y por ende retroalimentar el proceso. Debe ofrecer una muestra de los aspectos claves del problema y proveer una solución lo suficientemente simple para ser comprendida e implementada fácilmente. Se define una **lista de control de proyecto**.

- **Etapa de iteración.** Esta etapa involucra el rediseño e implementación de una tarea de la lista de control de proyecto, y el análisis de la versión más reciente del sistema. La meta del diseño e implementación de cualquier iteración es ser simple, directa y modular, para poder soportar el rediseño de la etapa o como una tarea añadida a la lista de control de proyecto. El código puede, en ciertos casos, representar la mayor fuente de documentación del sistema. El análisis de una iteración se basa en la retroalimentación del usuario y en análisis de la funcionalidad disponible.

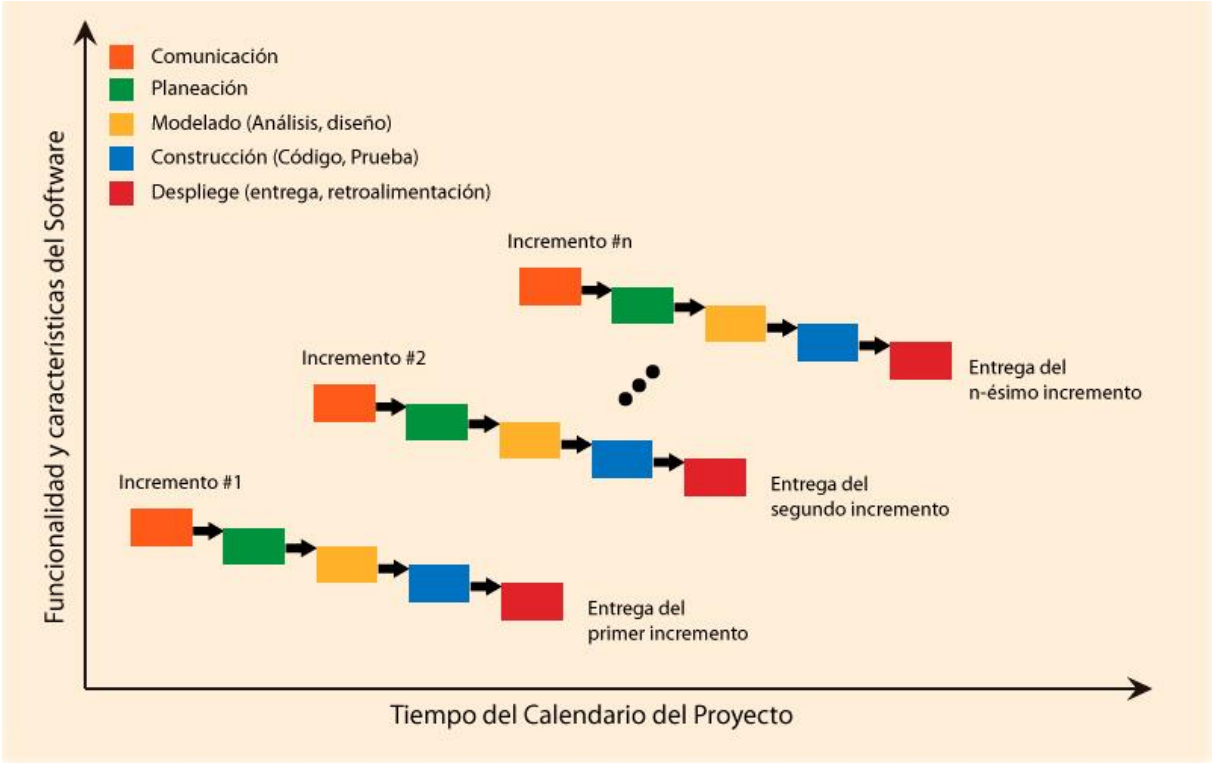

Figura 1.47: Metodología de desarrollo incremental. Fuente: (S. Pressman, 2006)

1.11.3 Justificación

Al tratarse este Trabajo Fin de Grado de un trabajo **teórico/experimental**, con una potente actividad de investigación asociada al mismo, se considera que lo más adecuado para el desarrollo del mismo es adoptar una estrategia incremental.

El hecho de adoptar esta estrategia se debe a la gran labor investigadora que es necesaria para poder desarrollar este trabajo. Nuestro objetivo es la segmentación en nubes de puntos. Así, en la primera iteración, se implementaría una versión básica de la técnica escogida. En iteraciones posteriores, el trabajo

consistiría en seguir estudiando técnicas y extraer información útil para intentar mejorar la alternativa propuesta en la primera iteración, así como mejorar en eficiencia y resolver problemas aparecidos en iteraciones anteriores.

El resultado de un trabajo teórico/experimental no es generar ningún **producto software**, sino hacer un análisis del estado del arte de la temática en cuestión, y experimentar con una de las alternativas estudiadas. Por tanto, es difícil establecer un ciclo de vida diferente al incremental, ya que la labor de investigadora parte desde una base mínima y se va construyendo con tiempo y estudio; no hay unos requerimientos mínimos que satisfacer pues una alternativa no puede implementarse a su mitad, pues no tendría sentido. Además, continuamente hay que analizar nuevas alternativas y rediseñar la propuesta, realizando nuevos experimentos. Es decir, pasar por todas las fases del ciclo de vida. Por otra parte, no existe un rol de cliente o usuario, ya que el propio usuario será el propio desarrollador que necesita de hacer una experimentación con la técnica propuesta para añadir al trabajo.

Así, en este caso, será quien escribe este proyecto quien haga a la vez de cliente y desarrollador.

1.12 Estimación del tamaño y esfuerzo

Ya que el presente proyecto es un TFT, no existen restricciones de tipo económico, sino de tipo temporal (un número aproximado de horas). Por consiguiente, los cálculos de tamaño del proyecto están supeditados el tiempo disponible. En cuanto al esfuerzo, se dispone de tan un solo efectivo (la persona autora del trabajo).

1.13 Planificación temporal

Para conseguir los objetivos del presente proyecto, es necesario establecer una planificación temporal que determine la tarea realizada en cada período de la duración total, como se observa en la Tabla 1.3. El carácter de esta planificación será orientativo, pues se trata de una estimación y los tiempos reales podrían variar.

Tarea a realizar	Duración estimada
Estudio y formación en Redes Neuronales	8 semanas
Estudio y formación en Nubes de puntos	2 semanas
Estudio del estado del arte	4 semanas
Desarrollo del software	4 semanas
Pruebas	2 semanas
Experimentación y recogida de resultados	3 semanas
Elaboración de la documentación	5 semanas
Total	28 semanas

Tabla 1.3: Planificación temporal del proyecto

Es importante señalar que la estrategia de desarrollo de software es incremental y, por tanto, no habrá completo aislamiento entre la fase de desarrollo de software y la fase de pruebas, se trata de dos fases que irán intercalándose por la naturaleza iterativa de la estrategia.

Para tener una mejor visión de la planificación temporal y su ajuste en cuanto a semanas, elaboramos un **Diagrama de *Gantt*** (véase Tabla 1.4, Tabla 1.5 y Tabla 1.6).

		Semana										
Tarea	**Duración**	1	2	3	4	5	6	7	8	9	10	11
Redes neuronales	8 sem.											
Nubes de puntos	2 sem.											
Estado del arte	4 sem.											
Desarrollo	4 sem.											
Pruebas	2 sem.											
Experimentación	3 sem.											
Documentación	5 sem.											

Tabla 1.4: Diagrama de Gantt (I)

	Semana													
Tarea	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Redes neuronales														
Nubes de puntos														
Estado del arte														
Desarrollo														
Pruebas														
Experimentación														
Documentación														

Tabla 1.5: Diagrama de Gantt (II)

	Semana			
Tarea	26	27	28	
Redes neuronales				
Nubes de puntos				
Estado del arte				
Desarrollo				
Pruebas				
Experimentación				
Documentación				

Tabla 1.6: Diagrama de Gantt (III)

1.14 Presupuesto

En esta sección se realiza el estudio de los costes económicos del proyecto. Aunque se trate de un trabajo teórico-experimental que no tiene como objetivo el desarrollo de un producto software, existen unos gastos asociados al mismo como consecuencia de los materiales (*hardware/software*) a utilizar, las fuentes de datos, etcétera; toda actividad de investigación necesita ser subvencionada para poder desarrollarse.

1.14.1 Hardware

El *hardware* requerido para el desarrollo del proyecto conlleva unos costes exigentes asociados, pues se trata de un software muy potente que permita realizar

entrenamiento de redes neuronales, además de la gestión de un volumen de información muy denso.

1.14.1.1 Desarrollo

Ordenador del laboratorio	
Componente	**Precio**
Intel Core i7 7700	314,99 €
Gigabyte NVIDIA GTX 1060	339,90 €
16 GB Memoria RAM (2x 8GB)	144,00 €
1 TB disco duro Seagate	39,65 €
Caja, refrigeración, etcétera	250,00 €
Monitor DELL x2	200,00 €
Total	1.288,54 €

Tabla 1.7: Estimación de costes del hardware de desarrollo (I)

Ordenador de casa	
Componente	**Precio**
iMac 21” (finales 2014)	1.500,00 €
Monitor SAMSUNG 19”	300,00 €
Total	1.800,00 €

Tabla 1.8: Estimación de costes del hardware de desarrollo (II)

Total *hardware* de desarrollo	
Equipo	**Precio**
Ordenador del laboratorio	1.288,54 €
Ordenador de casa	1.800,00 €
Total	3.088,54 €

Tabla 1.9: Estimación de costes del hardware de desarrollo (III)

1.14.1.2 Experimentación

Ordenador de experimentación	
Componente	**Precio**
Intel Core i7 8700K	379,90 €
NVIDIA Titan Xp	1299,00 €
64 GB Memoria RAM (8x 8GB)	576 €
Seagate IronWolf 4TB	128,89 €
Caja, refrigeración, etcétera	350,00 €
Total	2.733,79 €

Tabla 1.10: Estimación de costes del hardware de experimentación

1.14.1.3 Total

Coste total		
Equipo	**Precio**	**P. amortizado 7 meses**
Desarrollo	3.088,54 €	360,12 €
Experimentación	2.733,79 €	319,14 €
Total	5.822,33 €	679,26 €

Tabla 1.11: Estimación del coste total del hardware

Si el **coste total de los equipos es de 5.822,33 €**, y supóngase que el gasto en los mismos es una cantidad que se amortiza en 5 años, necesitarían pagarse $\frac{5.822{,}33€}{5\ años} = 1.164{,}46\ €/año$ para amortizar los equipos. Dado que el proyecto tiene una **duración total de 7 meses** (28 semanas), en ese período **se amortizaría una cantidad de** $\frac{\mathbf{1.164{,}46\ €/año}}{\mathbf{12\ meses/año}} * \mathbf{7\ meses} = \mathbf{679{,}26\ €}$**,** lo que supone una fracción del total de $\frac{679{,}26€}{5.822{,}33€} * 100 = \mathbf{11{,}66\%}$ **amortizado** de los equipos.

1.14.2 Software y datos

Todo el *software* utilizado es gratuito o bien se puede obtener de forma gratuita por adquirir un producto de la compañía fabricante (como es el caso de Apple o NVIDIA).

Por otra parte, los recursos de datos utilizados también son gratuitos y están disponibles en la web.

Software/Recurso	Precio
Ubuntu Linux 18.04	0,00 €
macOS Mojave	0,00 €
NVIDIA CUDA	0,00 €
***Framework* PyTorch**	0,00 €
PyCharm *Community Edition*	0,00 €
***Dataset* Semantic 3D**	0,00 €
GitHub Profesional (acceso con cuenta UJA)	0,00 €
Anaconda 3	0,00 €
Python 3.7 (resto de librerías)	0,00 €
Total	0,00 €

Tabla 1.12: Estimación de costes del software

No se calcula la amortización para el coste del *software* pues este es nulo.

1.14.3 Recursos humanos

Rol	Tarifa	Horas	Precio
Investigador	30€/hora	320 horas	9.600,00 €
Programador	22€/hora	50 horas	1.100,00 €
		Total	10.700,00 €

Tabla 1.13: Estimación de costes en recursos humanos

1.14.4 Costes indirectos

Los gastos de consumo eléctrico incluyen tanto el consumo de los equipos, como el gasto de aire acondicionado en la sala donde se encuentra el equipo de experimentación.

Concepto	Tarifa	Precio
Consumo eléctrico	40,00€/mes	70,00 €
Acceso a internet	30,00€/mes	210,00 €
Transporte al laboratorio y reuniones	80,00€/mes	560,00 €
Total		840,00 €

Tabla 1.14: Estimación de costes indirectos

1.14.5 Estimación total

Concepto	Precio
***Hardware* (p. amortizado en los 7 meses de duración)**	679,26 €
Software	0,00 €
Recursos humanos	10.700,00 €
Costes indirectos	840,00 €
Total	12.219,26 €

Tabla 1.15: Estimación total de costes

Dado que el proyecto tiene una duración de 7 meses, el gasto en cada mes es de**: 1.745,60€/mes.**

2 DESARROLLO

En esta sección se va a abordar todo el proceso de desarrollo del software necesario para poder llevar a cabo la experimentación conveniente con la técnica escogida. Es importante destacar que la propuesta descrita a continuación es el resultado de varias iteraciones de prueba y error, hasta conseguir una implementación estable y eficiente.

Como se menciona en la sección anterior (véase la sección 1.11.1), la estrategia de desarrollo de software se basa en un modelo incremental. Por tanto, el proceso completo de desarrollo se dividirá en iteraciones, en las que el resultado de cada una será un incremento sobre el software final. De tal modo, el software final dará paso a realizar una experimentación exhaustiva y emitir un juicio de evaluación con respecto a la técnica estudiada y propuesta.

2.1 Estrategia a desarrollar

La presente propuesta se basa en la **voxelización de entornos locales** de una nube de puntos para entrenar una red neuronal con el objeto de que aprenda a extraer patrones espaciales y con ello ser capaz de asignar una **etiqueta semántica** a cada punto (Huang & You, 2016). Para ello, los *grids* de vóxeles han de ser de un tamaño fijo, ya que es condición para toda red neuronal convolucional el hecho de que la **dimensión de entrada sea siempre la misma**.

Cabe destacar que esta propuesta no parte de una voxelización de la nube en el sentido estricto. Es decir, no se genera un único *grid* de vóxeles para todos los puntos de la nube, sino que este *grid* se generará en entornos locales. Se irán seleccionando puntos de la nube, sobre los cuales se generará un *grid* de vóxeles tomando provecho de la **información de sus vecinos**.

Así, teniendo un conjunto de *grids* de vóxeles de iguales dimensiones ya generado, es posible abordar el problema con el diseño de una red neuronal convolucional 3D que acepte *grids* de vóxeles de un tamaño prefijado como entrada, y sea capaz de **emitir una etiqueta para cada uno de estos *grids* de vóxeles**. De

este modo, los datos de entrenamiento se generarán a partir de nubes de puntos de entrenamiento ya segmentadas, donde cada punto tiene asociada una etiqueta semántica. Al tratarse de aprendizaje supervisado, no podría realizarse de no tener nubes de puntos etiquetadas.

Por otra parte, es necesario que el *dataset* sea lo más abundante posible, pues las redes neuronales necesitan una cantidad de datos abundante para poder entrenarse y extraer patrones correctamente. A diferencia de los humanos, por ejemplo, **en una red neuronal no es suficiente con ver solamente una cara de una taza para poder extraer patrones e identificar cualquier tipo de taza**. Principalmente porque se pierde el efecto de la profundidad. Las redes necesitan tener todas las perspectivas posibles para el objeto o superficie en cuestión, de lo contrario, no podrán extraer patrones adecuados.

En el caso de los modelos 3D, como es el caso, **las redes necesitarán ver tantas perspectivas como sea posible**, para que el entrenamiento sea correcto y la red sea capaz de extraer patrones de un trozo de realidad con una maximización de la significancia.

Las redes convolucionales 3D se idearon principalmente con el objetivo de extraer patrones en datos de vídeo. Décadas después, numerosos grupos de investigación decidieron aplicarlas en otras áreas para resolver otra problemática diferente, por su habilidad para la representación de la componente tridimensional de la realidad. Una red de este tipo es capaz de aceptar como entrada *grids* de vóxeles de la misma forma en la que una red neuronal convolucional **tradicional** (2D) es capaz de consumir una matriz de píxeles (en el caso de imágenes).

2.1.1 Preprocesamiento de datos

Una de las fases más cruciales (en toda aplicación de aprendizaje automático) es realizar un correcto tratamiento de los datos de manera que los mismos se encuentren de la forma más óptima posible y sea más fácil para la red entrenarse.

Las nubes de puntos utilizadas en ámbitos como la arquitectura, la planificación urbana, la gestión del territorio o arqueología, constituyen conjuntos de datos muy densos, especialmente cuando estas nubes proceden de escaneados LiDAR (ALS/TLS). Para poder ser tratados de una manera eficiente con este enfoque de red neuronal, es necesario establecer una **estrategia de muestreo** que nos permita **reducir este volumen de puntos**, pero siempre buscando un **compromiso entre la maximización del volumen espacial a tratar, y la minimización de la cantidad de puntos a procesar**. Es decir, necesitamos **obtener nubes lo más descriptivas posible utilizando el menor número de puntos**.

Por otra parte, **todo algoritmo de aprendizaje necesita de un conjunto de datos balanceado para entrenarse correctamente**. Es decir, el número de elementos de cada clase debe ser el mismo. De lo contrario, el modelo entrenado tenderá a predecir siempre la misma clase para todos los objetos, siendo esta clase la correspondiente a la que tiene el mayor número de elementos. En nuestro caso, deberemos escoger una estrategia para obtener un *dataset* balanceado, con el mismo número de puntos en cada categoría semántica.

2.1.1.1 Muestreo

Cuando se trata de trabajar con nubes de puntos muy densas (de varios cientos de millones de puntos), se vuelve muy difícil para una red procesar este volumen de información. Además, no solamente es un problema para la red, sino que previamente es necesaria la voxelización en entornos locales, resultando en modelos que ya son voluminosos por naturaleza y que también provocarían serios problemas de rendimiento. Por ende, como ya se ha dicho, el objetivo es obtener una nueva nube lo más representativa posible sobre la original, extrayendo los puntos más informativos de la misma.

Para realizar este muestreo debe establecerse un nivel de precisión, P. Será la distancia media entre cualesquiera dos puntos vecinos de la nueva nube muestreada. La mínima caja envolvente se divide en un grid o espacio discreto de $I \times J \times K$ celdas. Cada una de las celdas encerrará un conjunto arbitrario de puntos, de los cuales se selecciona uno como representante (aleatoriamente, para

evitar un sesgo del método), que pasará a formar parte de la nueva nube. La decisión sobre el nivel de precisión adecuado será crucial, pues si P es demasiado grande, se perderá mucha información relevante y la nube muestreada será poco descriptiva. En cambio, si P es pequeña, se habrá realizado un sobremuestreo y se tendrá más información de la necesaria, perjudicando así el rendimiento del sistema.

2.1.1.2 Balanceo del *dataset*

Como ya se ha dicho antes, todo algoritmo de aprendizaje necesita de un conjunto de datos balanceado sobre el que entrenarse. La mayoría de *datasets* para aprendizaje automático no están balanceados, siendo desigual el número de elementos en cada categoría.

En el caso de las nubes de puntos, este problema se acentúa. Por ejemplo, en nubes generadas mediante LiDAR, normalmente se utilizará el escáner en escenas rurales o urbanas. Si se trata de escenas **rurales**, la mayoría de puntos estarán concentrados en terreno, vegetación, etcétera, desfavoreciendo a otras categorías como podrían ser rocas o pequeñas edificaciones en el campo. Si se trata de escenas **urbanas**, la mayoría de puntos corresponderán a edificios o pavimento, desfavoreciendo a otras categorías como vehículos. Es necesario establecer un compromiso entre ellas para que ninguna categoría sea favorecida sobre las demás.

Nuestra propuesta consiste en muestrear aleatoriamente sin reemplazamiento N puntos en cada categoría, siendo N el número de puntos de la categoría más desfavorecida. De este modo, se garantiza que todas las categorías tendrán el mismo número de puntos para la generación de las voxelizaciones locales, favoreciendo el balanceo. El hecho de que el muestreo sea aleatorio sin reemplazamiento garantiza que el espacio quedará cubierto prácticamente en su totalidad. Es decir, habrá puntos de cualquier región de la nube, y no solamente de una región aislada.

Estos puntos seleccionados serán los centros de las regiones voxelizadas, con las que se alimentará la red neuronal. Si bien cabe destacar que, aunque se haga una selección de puntos para favorecer el balanceo, la generación del *grid* de

vóxeles local se hará teniendo en cuenta todos los puntos de la nube, y no únicamente los puntos seleccionados. De esta manera, se tendrán *grid* de vóxeles lo más informativos posible.

2.1.1.3 Aumento de *dataset*

Uno de los problemas que presentan las redes neuronales en general, y las redes convolucionales en particular, es que necesitan un volumen de datos lo más completo posible y desde **todas las perspectivas posibles** para poder extraer patrones correctamente. Como ya se ha ejemplificado antes, una red convolucional es incapaz de reconocer cualquier tipo de taza entrenándola con una sola perspectiva de un solo tipo de taza. Estas redes necesitan entrenarse con todas las perspectivas y casuísticas posibles.

Para compensar esta deficiencia, los *frameworks* de aprendizaje profundo suelen ir acompañados de operaciones que permiten aumentar el *dataset* en cuanto a aplicar pequeñas transformaciones sobre los elementos y así tener diferentes perspectivas. Por ejemplo, sobre una imagen, suelen hacerse algunos recortes, variar un poco el color, voltear la imagen lateralmente, etcétera.

En nuestro caso, se decide optar por una estrategia diferente y aplicar 12 transformaciones sucesivas acumuladas de rotación sobre cada entorno local tras hacer la extracción de vecinos para un punto seleccionado. Este conjunto de vecinos estará formado por todos los puntos que encerrará el *grid* de vóxeles, viniendo determinada esta medida por el tamaño de vóxel y el número de vóxeles en el *grid*. Así, sobre este conjunto (que incluye al punto generador) se aplican 12 rotaciones acumuladas de 30 grados cada una alrededor del eje Y en el sistema de coordenadas de mano derecha. Esto enseñará a la red diferentes puntos de vista de cada zona espacial, potenciando el aprendizaje.

2.1.2 Voxelizaciones locales

Uno de los pilares fundamentales de nuestra propuesta consiste en la generación de voxelizaciones locales de tamaño fijo. Dada una nube de puntos definida sobre el espacio métrico, donde cada punto tiene coordenadas (x, y, z), se seleccionan N puntos con los criterios presentados en la sección 2.1.1.2, y se realiza

una voxelización local centrada en cada uno de dichos puntos. Cabe destacar que no es posible realizar la operación para toda la nube y posteriormente ir seleccionando subregiones para ganar en rendimiento, pues las regiones transformadas en torno a cada punto seleccionado no están perfectamente alineadas. Para conseguir esto, la distancia entre los puntos vecinos debería ser uniforme en cada una de las tres dimensiones para toda la nube.

Para extraer patrones espaciales correctamente, cada punto analizado debe estar acompañado de la información de sus vecinos más cercanos. Por tanto, es necesario definir el tamaño de las voxelizaciones locales, que será de $N \times N \times N$ vóxeles, y también la dimensión de cada vóxel, t, que determina la porción del espacio métrico que ocupa. Así, sea P un punto de los seleccionados, con coordenadas (x, y, z), el grid generado tengrá una caja envolvente de $[x - \frac{N*t}{2}, x + \frac{N*t}{2}] \times [y - \frac{N*t}{2}, y + \frac{N*t}{2}] \times [z - \frac{N*t}{2}, z + \frac{N*t}{2}]$ unidades espaciales (normalmente se dará en metros). La voxelización será diferente en etapas de entrenamiento y test.

Un *grid* de vóxeles es una estructura de datos sencilla en la que se tiene un *array* tridimensional, y cada celda de ese *array* se corresponde con un vóxel diferente. Aunque existen diferentes formas de asignar un valor a una celda (ocupación, densidad, etcétera), se opta por la más sencilla pues es suficiente para lo que se pretende conseguir. Por tanto, independientemente del número de puntos encerrados dentro de un vóxel:

$$grid(i,j,k) = \begin{cases} 1, & hay\ algún\ punto\ en\ el\ vóxel\ (i,j,k) \\ 0, & e.o.c. \end{cases} \quad (2.1)$$

Sean i, j, k las coordenadas discretas normalizadas al subespacio que genera el grid de vóxeles definidas en el rango $[0, N) \times [0, N) \times [0, N)$.

Así, este enfoque pretende representar la morfología de los objetos, prescindiendo que cualquier otra información como el color, el vector normal en cada punto, etcétera, con el propósito de que la red se encargue de estimar cuál es la

medida con la que mejor se aproxima cada patrón, y con esto segmentar cada punto a partir de la representación mediante *grids* de vóxeles.

2.1.2.1 Distinción entre entrenamiento, desarrollo y test

Durante el **entrenamiento**, es necesario tener en cuenta la etiqueta semántica de los puntos con los que se está trabajando. Para cada punto seleccionado se genera un *grid* de vóxeles con las características especificadas anteriormente. Cada *grid* será a su vez la representación de un punto en concreto y la caracterización de este punto en un entorno local. Es decir, si se trata de suelo, la red aprenderá que un punto pertenece al suelo porque normalmente la mayoría de puntos vecinos están en el mismo plano y pertenecerán a suelo.

Por tanto, un *grid* se etiquetará con una única etiqueta, que se corresponderá con la mayoritaria entre los puntos que estén a menos de una distancia de $\frac{t}{2}$ del punto a partir del cuál se generó el vóxel.

Por otra parte, se distingue la voxelización local durante el **test**. En este caso no se necesita conocer la etiqueta de cada punto a priori, pues es trabajo de la red **inferir** cuál es la etiqueta de cada punto. Con el test se consigue medir la calidad de un algoritmo una vez que ha sido perfectamente entrenado y validado.

La entrada a la fase de test es una nube de puntos sin etiquetar. En este caso no será necesario realizar muestreo pues los *grid* de vóxeles no se generarán a partir de cada punto, aunque sí se comparte cierto proceso con el muestreo. En primer lugar, se calcula la Mínima Caja Envolvente (MBB, *Minimal Bounding Box*) para la nube de entrada, con tantos puntos como contenga originalmente. A continuación, esta caja es dividida por la dimensión del vóxel, que ha de ser la misma que se utilizó en el entrenamiento, resultando en un espacio discreto de $I \times J \times K$ celdas. Cada celda encerrará un conjunto arbitrario de puntos. Para cada una de las celdas en las que hay algún punto, se interpola el punto central (realmente este punto no tiene por qué existir), y tras hacer esto para todas las celdas en las que hay algún punto, se procede a hacer las voxelizaciones locales de

la misma manera que en la fase anterior, con la diferencia de que en esta etapa no se asigna ninguna etiqueta pues a priori se desconoce.

Una vez generados todos los *grids*, se dan como entrada a la red, y ésta generará exactamente una etiqueta para cada *grid*. Tras esto, todos los puntos que están en una celda del espacio discreto original se etiquetan con la misma clase que emitió la red para el *grid* generado a partir del punto central interpolado de esa celda. De este modo, para nubes con varios cientos de millones de puntos, no es necesario generar un *grid* para cada punto, pues de hecho varios puntos muy cercanos compartirán la etiqueta.

De este modo queda descrito el proceso de voxelización local. Un detalle importante a mencionar es la elección de la **dimensión del vóxel** y la **precisión**. Lo ideal es escoger la misma medida para ambos, si bien la dimensión del vóxel es un aspecto crucial ya que determinará la representatividad de los objetos. Por ejemplo, si queremos que una papelera pueda ser descrita mediante vóxeles, suponiendo unas dimensiones de $0{,}5\,m * 0{,}3m * 0{,}3m$ de alto, ancho y largo para la misma, la dimensión del vóxel no deberá superar los $0{,}15m$, pues cualquier dimensión más grande hará que la papelera sea representada mediante vóxeles de una manera equivalente a otras formas, como por ejemplo una fuente de agua.

2.1.3 Arquitectura de la red

La propuesta de arquitectura de este enfoque está inspirada en el éxito de LeNet-5 para clasificación de imágenes de caracteres manuscritos de naturaleza monocanal. Al tratarse igualmente de un solo canal, aunque 3D, para representar la morfología de la nube, se decide optar por esta arquitectura, aunque cambiando las componentes convolucionales de 2D a 3D. La propuesta de arquitectura está ilustrada en la Figura 2.1.

La red parte de dos capas de convolución 3D, seguida cada una de ellas por una capa de *max-pooling* 3D para resumir características, culminando con una red *fully-connected* (tipo perceptrón multicapa) para poder segmentar semánticamente haciendo uso de las características extraídas en capas anteriores. Este proyecto se inspira en las arquitecturas habituales para clasificación de imágenes, extendidas en

este caso a tres dimensiones. Como ya se dijo antes, la convolución 3D se ideó inicialmente para extraer patrones en datos de vídeo, aunque en este caso se utilizará este enfoque para extraer patrones espaciales de los distintos *grids* de vóxeles con los que se servirá de entrada a la red neuronal. Mientras que las capas de convolución se encargan de extraer características, las capas de *pooling* se encargan de reducir la dimensionalidad de estas características, simplificándolas y haciendo más fácil para una red la concordancia entre patrones para diferentes datos de entrada.

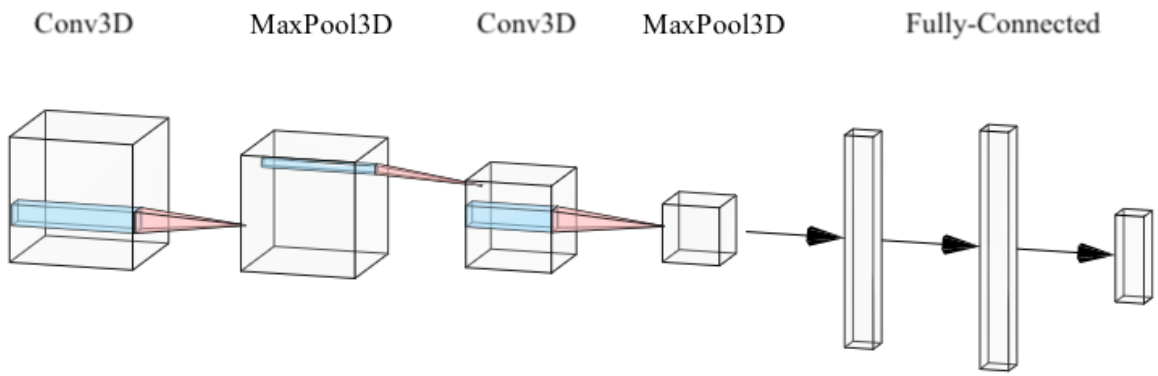

Figura 2.1: Arquitectura de Red Neuronal Convolucional 3D propuesta

Siguiendo el paradigma anterior se expone el siguiente ejemplo: con un *grid* de vóxeles de 20^3 vóxeles, tomando un número de 20 filtros tanto en la primera capa de convolución como en la segunda, un tamaño para el *kernel* de 5^3 y un *stride* de 1 en las dos capas; y un tamaño de *kernel* de 2^3, un *stride* de 2 en las capas de *max-pooling*, llegamos a la capa *fully-connected* con un grid de dimensiones $2 \times 2 \times 2 \times 20$ filtros, lo que resulta en un vector de características de tamaño 160. La parte *fully-connected* de la red está compuesta por 3 capas: la capa de entrada, la capa oculta, y la capa de salida, con 160, 300 y M unidades respectivamente. La capa de entrada ha de configurarse según la dimensión del *grid* que llega a ella, mientras que M en la capa de salida representa el número de clases distinguibles en el problema, y entre las que la red tendrá que emitir una etiqueta.

2.2 Iteraciones

La manera en la que va a describirse cada iteración pretende: realizar un análisis de lo que debe conseguirse con el software, diseñar una propuesta de desarrollo, implementar el software diseñado, y realizar pruebas, extrayendo conclusiones de los cambios que deberían hacerse en cada iteración.

Como ya se comentó en la sección anterior, la solución antes propuesta es el resultado de una serie de iteraciones de prueba y error de manera que, desde el comienzo de esta sección, la solución que va a ir proponiéndose distará de lo que está especificado arriba (véase sección 2.1) hasta haber avanzado algunas iteraciones.

2.2.1 Primera Iteración

2.2.1.1 Dataset

En esta primera iteración se pretende conseguir una versión funcional del sistema propuesto. Para ello, el primer paso a dar consiste en la obtención del *dataset* de nubes de puntos etiquetadas. Se proponen varias alternativas: utilizar datos LiDAR en formato .las/.laz, donde existen librerías específicas para poder cargar las nubes de puntos y procesarlas, como libLAS; o utilizar *datasets* como *Oakland Urban Dataset*. El segundo caso no sirve, pues no está etiquetado a nivel de punto, sino que está fragmentado en una serie de segmentos con una etiqueta semántica global. Así, se utilizarán los *datasets* de LiDAR disponibles, como el correspondiente a la Figura 2.2. Por otra parte, se completará la voxelización en entornos locales, de manera que pueda establecerse el diseño de la red neuronal tal y como está descrita.

Figura 2.2: Nube de puntos de la ciudad de Jaén generada mediante LiDAR (ALS)

En primer lugar, se procede a la implementación del código para la voxelización. Para ello, dado que el lenguaje de programación es Python, se crea un entorno virtual en Anaconda 3 y se instalan todos los paquetes necesarios, incluyendo NumPy (https://www.numpy.org/, Figura 2.3) y libLAS (https://liblas.org/), para el trabajo con *datasets* LiDAR. La instalación de libLAS debe hacerse tanto a través de *pip*, para el *wrapper* de Python, como a través del gestor de paquetes de Ubuntu/macOS (MacPorts, https://www.macports.org/) con el comando *sudo apt install liblas/sudo port install liblas* respectivamente.

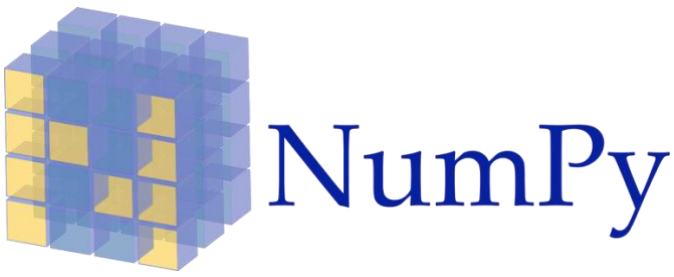

Figura 2.3: Logotipo de NumPy

2.2.1.2 Voxelización

Es necesario definir el número de vóxeles que compondrán el *grid* así como la dimensión de vóxel. (Huang & You, 2016) Se toma un número de 20^3 vóxeles, con un tamaño de vóxel de 0,2 metros.

En un primer momento, para favorecer el balanceo, se toma la decisión de extraer de cada categoría tantos puntos como haya en la categoría minoritaria (véase 2.1.1.2). Serán estos puntos los que darán paso a la realización de las diferentes voxelizaciones locales. El procedimiento es el siguiente:

Para cada punto seleccionado, se obtienen sus vecinos: los puntos que encierra el grid de vóxeles dadas sus dimensiones (véase Figura 2.4). Se inicializa el *grid* de vóxeles como un array tridimensional de dimensiones (20, 20, 20) de NumPy, inicializado a 0.

Por cada uno de los vecinos, se calcula el vóxel que ocupa dentro del *grid* y asignamos a este un 1. Así, suponiendo que el punto cae en la celda (i, j, k) del grid, se asigna: $grid(i, j, k) = 1$. A la vez que se calcula el vóxel correspondiente a cada punto vecino, es necesario comprobar si este punto está a menos de una distancia de $\frac{t}{2}$ del punto generador, siendo t la dimensión del vóxel. En tal caso, se almacena el vecino en una lista para más tarde tenerlo en cuenta al momento de asignar una etiqueta al *grid*. Tras operar sobre todos los vecinos, el resultado es un *grid* de vóxeles de un tamaño prefijado, como se observa en la Figura 2.5.

Figura 2.4: Entorno local de puntos

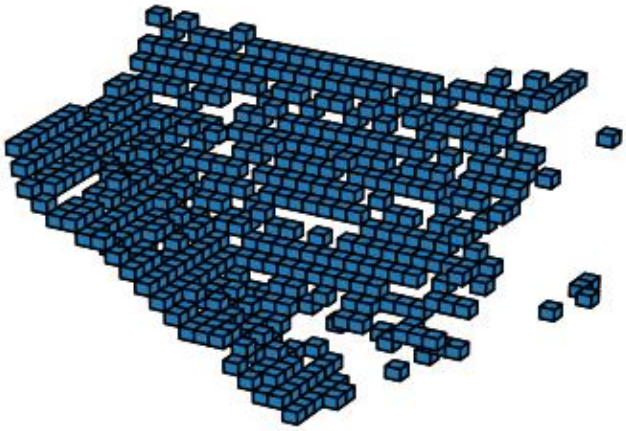

Figura 2.5: Voxelización de un entorno local de puntos

2.2.1.3 Asignación de etiquetas

Tras generar el *grid*, la siguiente tarea consiste en asignar una etiqueta. La etiqueta a asignar se corresponderá con la mayoritaria de entre todos los puntos que están a menos de una distancia determinada del punto central (inclusive). En caso de empate, se escoge la etiqueta del punto generador.

Para que las etiquetas puedan ser aceptadas por la red deben estar en el rango $[0, N)$, siendo N el número de clases. Por tanto, una vez ha sido seleccionada la etiqueta, se resta 1 unidad para pasarla al rango deseado, pues el rango original es $[1, 8]$ y se pretende obtener $[0, 7]$.

2.2.1.4 Almacenamiento de ficheros

Habiendo completado el proceso, se almacena el *array* de NumPy en un fichero utilizando el método *np.savez()*. De esta manera, cada fichero ocupa un total de 64KBytes de disco, además del *inode* (por tratarse de sistemas que provienen de UNIX) correspondiente para el fichero. A cada fichero se le asigna un número identificativo y se almacena con extensión ***.voxel***. Por otra parte, se tiene un fichero *.csv* en el que se anotan en forma de pares la ruta del fichero almacenado y la clase a la que pertenece, uno por línea. De manera que el fichero *.voxel* tendría una estructura (véase Tabla 2.1):

Identificador de fichero	Etiqueta
./grids/0.voxel	0
…	…
./grids/M.voxel	5

Tabla 2.1: Estructura del fichero de etiquetas en iteración 1

M representa al número total de grids generados. El fichero *.csv* está separado por el carácter **|** (alt + 1).

2.2.1.5 Conclusiones

Si se tiene un total de 1 millón de puntos, como será lo habitual e incluso más en datos de entrenamiento, se generaría 1 millón de ficheros, sumando un total de 64.000.000 KB = 64GB. El volumen de información es muy grande y difícilmente manejable, teniendo en cuenta además de que en la mayoría de celdas de los grid de vóxeles habrá un 0, pues no están ocupados por ningún punto. Es necesario diseñar una estrategia para reducir el volumen de datos.

Con el volumen de datos actual es difícil establecer un diseño de red neuronal, pues existe una limitación en escrituras a disco por la cantidad de información que se genera durante la voxelización, y el entrenamiento se vería

gravemente penalizado debido a que la GPU estaría continuamente nuevos datos que tendrían que leerse de disco. Por otra parte, se extrae la conclusión de que **los datos LiDAR con los que se está trabajando no son adecuados para el entrenamiento debido a la inexactitud del escáner en el momento de asignar una etiqueta a los puntos** y la irregularidad de los mismos, de manera que solo se tienen algunas nubes de puntos LiDAR de utilidad, no siendo suficiente para entrenamiento y posterior validación de una red neuronal. Por otra parte, los *datasets* LiDAR generados mediante ALS (sensor aéreo) suelen proporcionar una distancia entre puntos de unos 0,3 m. Esto limita mucho la precisión a la hora de representar objetos como **tendido eléctrico**, por ejemplo. Aunque esta categoría no vaya a tratarse, la naturaleza dispersa de los puntos ya introduce un sesgo no apropiado en una de las fases primordiales del método, la generación del conjunto de entrenamiento.

Del mismo modo, se desperdicia mucho tiempo cada vez que se consultan los vecinos para un punto, dado que es necesario explorar todos los puntos de la nube y quedarse con aquellos que permanezcan dentro del espacio que encierra el *grid*, resultando en un orden de complejidad de $O(n)$, donde n es el número de puntos de la nube.

Antes de continuar con el desarrollo de la red, es necesario trabajar sobre una estrategia que almacene un volumen de datos menor y una optimización en las consultas de vecinos, a la vez que es necesario utilizar un *dataset* diferente, con el etiquetado de puntos correcto y lo suficientemente denso.

2.2.2 Segunda iteración

2.2.2.1 Nuevo *dataset*

Tomando como realimentación lo acontecido en la iteración anterior, el primer problema a resolver consiste en la selección de un *dataset* adecuado para resolver la tarea. Debe estar etiquetado a nivel de punto entre las diferentes categorías semánticas: Semantic 3D (descrito en 1.9.1).

2.2.2.2 Muestreo

El nuevo *dataset* cuenta con **una gran densidad de puntos**, del orden de pocos milímetros entre puntos, y a diferencia del utilizado en la iteración anterior está generado mediante LiDAR terrestre (TLS). Como ya fue descrito en 2.1.2, la elección del tamaño de vóxel es muy importante, pues toda variación dentro del mismo se pierde. Así, si se escoge un tamaño de vóxel de 0,2 m y la distancia entre puntos es menor, no se aprovechará esta mayor densidad pues la diferencia residirá en que varios puntos estarán dentro de un mismo vóxel, y la variación entre los mismos se pierde.

Es necesario aplicar una estrategia de muestreo sobre las nubes originales tal y como se describe en 2.1.1.1. Para evitar que el volumen de puntos supere la necesidad, y con el objeto de reducir en la medida de lo posible la densidad de puntos de la nube sin perder información, lo más adecuado es establecer un **compromiso entre la precisión de muestreo y el tamaño de vóxel**, siendo estos la misma medida. De esta forma, en cada vóxel habrá a lo sumo 1 punto (o 2 puntos ya que el muestreo no es uniforme a nivel de puntos centrales de la región, sino aleatorio), aprovechando al máximo el carácter informativo de la nube con el mínimo número de elementos posible. Como analogía, se puede pensar en un vídeo con resolución 4K visualizándose sobre un monitor de 800x600 píxeles. No se apreciará el nivel de detalle del vídeo y además es muy voluminoso; sería suficiente con un vídeo en calidad 720p o incluso menor.

Tomando una precisión de 0,2m tanto para la precisión como para el tamaño de vóxel, se alcanza una disminución de 280 millones a 0,43 millones en el caso de la nube más densa, lo que representa tan solo un 0,1% de la nube original, sin pérdida de información.

2.2.2.3 Mejora en la búsqueda de vecinos

Por otra parte, la operación de extracción de vecinos para un punto una vez que este ha sido seleccionado es ineficiente, del orden de $O(n)$ (con n el número de puntos). Es necesario utilizar algún tipo de **estructura espacial** que permita acelerar estas búsquedas, como lo son el **octree** o **kd-tree**. En un principio iban a utilizarse *arrays* dispersos (*sparse*) de SciPy (https://www.scipy.org/, Figura 2.6), pero la

funcionalidad necesaria para el presente proyecto es mínima, y es innecesario importar otra librería más en el código. En este caso se opta por un *grid* disperso construido mediante diccionarios en Python por simplicidad en el código.

Figura 2.6: Logotipo de SciPy

Se calcula la caja envolvente mínima (MBB) para la nube, y es dividida en regiones de tamaño t, la dimensión del vóxel. De nuevo, se distribuyen los distintos puntos por espacio discreto generado como si de una **malla regular** se tratase. Esta estructura se mantiene en memoria. Todo el procesamiento es manejado por una clase *GridDisperso*. Existe un método que recibe como entrada las coordenadas de un punto, el número de vóxeles en el *grid*, y el tamaño de vóxel, y de un modo eficiente es capaz de acceder solamente a las celdas que intersectan con el espacio que encierra el vóxel y devolver únicamente los puntos vecinos sin necesidad de explorar todos los puntos. Mediante esta operación, se mejora la eficiencia del orden de complejidad $O(n)$ a $O(1)$.

2.2.2.4 Optimización del almacenamiento

Otro de los problemas detectados en la primera iteración es el volumen de almacenamiento que se desperdicia al almacenar cada *grid* de vóxeles como un fichero independiente. Tomando un tamaño para el *grid* de 20^3 vóxeles, se genera un *array* de NumPy con 8000 posiciones. Para que el almacenamiento fuese de provecho, la mayoría de *grids* deberían tener ocupadas esas 8000 posiciones en la mayoría de los casos o al menos una fracción representativa, lo cual no será habitual pues cada *grid* de vóxeles representará un fragmento de la superficie muestreada en la nube.

Una nube de puntos LiDAR es una representación 2,5D, es decir, algo así como la corteza de la realidad representada. Por tanto, en altura (eje Z), normalmente se ocuparán pocas posiciones del *array*, mientras que en los ejes X e

Y (paralelos a la superficie) normalmente siempre habrá vóxeles. Como se aprecia en la Figura 2.5, mientras que en anchura y profundidad siempre existen vóxeles, en altura hay mucho espacio que no se utiliza.

El problema aquí presente es el mismo que el precursor de las matrices dispersas; siendo más concretos, se trata de encontrar una representación menos voluminosa pues la mayoría de celdas de una matriz van a contener un valor 0. Por tanto, aprovechando la idea del *grid* disperso utilizado para optimizar las búsquedas, se implementa cada *grid* de vóxeles como si de un *grid* disperso se tratase, almacenando únicamente las posiciones (i, j, k) del grid en las que hay un vóxel. Un *grid* es fácilmente representado mediante un diccionario (*dict)* de Python.

Suponiendo que se está representando una superficie de terreno, en el cuál todos los puntos son coplanarios y el vector normal es paralelo al eje Z, todos los puntos se representarán en una única fila de vóxeles (véase Figura 2.7), ocupando un total de 400 vóxeles si el tamaño del *grid* es de 20^3, mientras que el tamaño total del *array* es de 8000 posiciones. Con esta alternativa, se reduce el volumen de almacenamiento en un 2000%.

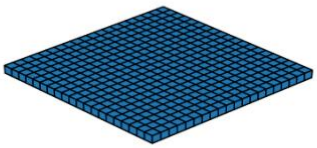

Figura 2.7: Ejemplo de voxelización para suelo

Tanto la estructura del fichero *.csv* como la filosofía de almacenamiento es la misma (1 *grid* de vóxeles → 1 fichero), a diferencia de que el método para guardar el *grid* en disco se utiliza la librería *Pickle* de Python para guardar el diccionario como un objeto binario y así acelerar tanto el almacenamiento como la posterior carga.

2.2.2.5 Conclusiones

Una vez resueltos todos los problemas surgidos durante la primera iteración, aparece la necesidad de realizar un aumento de *dataset* para solucionar el problema inherente a las redes convolucionales: la **gran cantidad de datos de entrenamiento** que requieren; de manera que una red pueda reconocer que un punto pertenece a un edificio con independencia de la rotación de la pared que se forma en un entorno de vecindario.

Por otra parte, también es necesaria la división del conjunto de **entrenamiento** en **entrenamiento** y **desarrollo (validación)** para poder evaluar la calidad del modelo entrenado y poder ajustar correctamente los hiperparámetros.

2.2.3 Tercera iteración

En esta tercera iteración pretenden resolverse tanto el problema del aumento de datos como la división entre conjuntos de entrenamiento y desarrollo (validación).

2.2.3.1 Aumento de datos

Para resolver el problema del aumento de datos se toman las consideraciones de VoxNet (Maturana & Scherer, 2015) tal y como se ha descrito en la sección 2.1.1.3. La idea consiste en aplicar 12 rotaciones sucesivas acumuladas de 30 grados cada una, de forma que el conjunto de vecinos rote sobre el eje perpendicular a la superficie terrestre.

El modo de conseguir esto parte de aplicar una transformación geométrica de **rotación** en el sentido tradicional de informática gráfica. Por tanto, los pasos son: una vez se dispone tanto del punto que generará el *grid* como de su entorno, se traslada el entorno local al origen de manera que el punto generador quede situado en el origen de coordenadas, se aplica la **rotación**, y seguidamente se realiza la voxelización. En informática gráfica sería necesario volver a llevar los puntos a su lugar de origen tras la rotación, pero el objetivo no es *rendering*, sino una voxelización que discretizará la ubicación de los puntos en un espacio discreto de $N \times N \times N$ dimensiones. Este procedimiento de aplicar la rotación se repite 12

veces, mientras que la traslación solo se hace al principio. El hecho de aplicar la traslación es necesario para que la rotación no deforme la nube.

Para realizar la rotación de una manera eficiente, se genera una matriz de NumPy con las coordenadas de todos los puntos en el entorno local, de manera que la transformación de rotación pueda aplicarse sobre todos los puntos a la vez y ganar en eficiencia. El aspecto de la matriz de rotación, R, aplicada sobre los puntos una vez se han trasladado al origen es el siguiente:

$$R = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

donde α será el ángulo de rotación, que variará entre 0 y 360º con un incremento de 30º. De esta manera, se consiguen obtener 12 perspectivas diferentes para cualquier fragmento del espacio que vaya a voxelizarse, incrementando la potencia de la red para extraer patrones concretos en cuanto a vecindario que rodea a un punto (véase Figura 2.8).

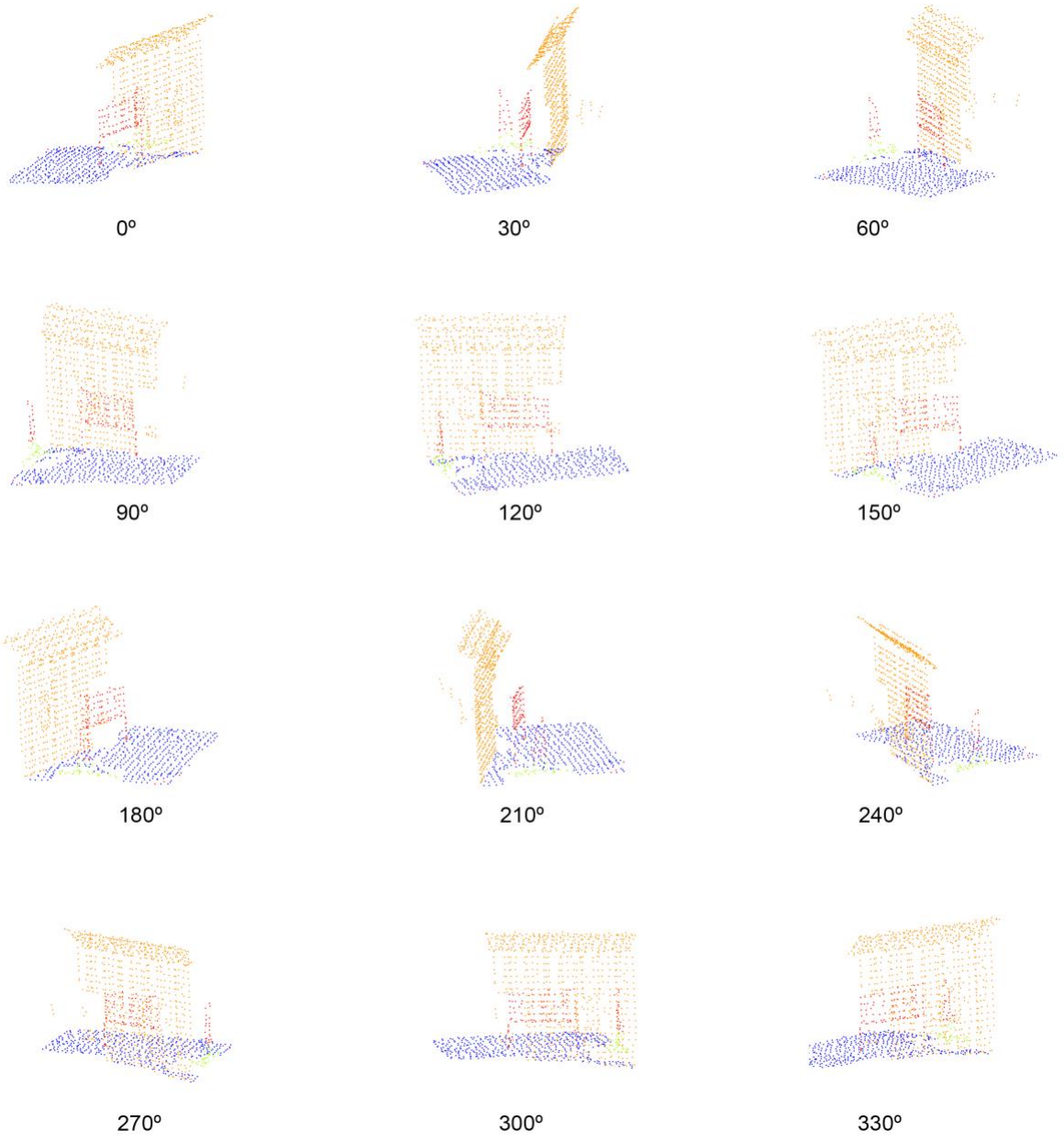

Figura 2.8: Aplicación de 12 rotaciones acumuladas de 30º sobre un entorno local

2.2.3.2 División entre conjuntos de entrenamiento y desarrollo: *Hold out*

Una vez se dispone de un marco de trabajo adecuado para la realización de los experimentos, así como un aprovechamiento eficiente del espacio en disco para los *grids* de vóxeles generados, es necesario escoger una estrategia que permita tanto **entrenar** correctamente un modelo predictor, como **evaluar** la calidad del conocimiento extraído durante el entrenamiento.

De entre todas las estrategias disponibles para entrenamiento/validación, se opta por ***Hold out*** por tratarse de conjunto de datos voluminoso, con abundancia de ejemplos tanto de entrenamiento como de validación. Se trata de un tipo de **validación a una sola vuelta**: se divide el conjunto de datos en conjuntos disjuntos de entrenamiento y validación, con una proporción de ejemplos desigual (entrenamiento > validación). El conjunto de entrenamiento está formado por ejemplos muestreados sin reemplazamiento del conjunto original, mientras que el conjunto de validación está formado por todos los ejemplos restantes en el conjunto original que no han sido seleccionados.

En la clase *Voxelize* existe un método para generar el conjunto de entrenamiento tomando como entrada una nube de puntos. Dentro de este método, hay un bucle que itera sobre todos los puntos seleccionados, entre las diferentes categorías a las que pertenecen, cumpliendo las condiciones del balanceo, etcétera. El método se encarga de repartir un porcentaje α de cada categoría en el conjunto de entrenamiento, y un porcentaje $1 - \alpha$ para el conjunto de validación. Destacar que los conjuntos son disjuntos, esto es, los ejemplos que están en un conjunto no están en el otro y viceversa.

Sobre los ejemplos destinados a formar parte del conjunto de entrenamiento se aplica la estrategia de aumento de datos (12 rotaciones) para incrementar el campo de visión de la red en cuanto extracción de patrones. De esta forma, el *dataset* de entrenamiento se vuelve 12 veces más voluminoso de lo previsto.

2.2.3.3 Conclusiones

Tras resolver las cuestiones presentes en la iteración anterior, aparece un nuevo problema relacionado con el almacenamiento de ficheros. El aumento de *dataset* para el entrenamiento en un factor de 12 provoca que el número de ficheros se multiplique por dicha cantidad en consecuencia. En todo sistema operativo, la operación de indexación de ficheros está altamente penalizada cuando el número de ficheros a tratar dentro de un directorio se hace muy grande.

Haciendo pruebas con el nuevo *framework*, llegaba a alcanzarse una cantidad de 1,5 millones de ficheros (*grids* de vóxeles) para entrenamiento. No solamente es

difícilmente manejable para el sistema operativo en el momento de listar el contenido del directorio y manipularlo, sino que cada vez que va a procesarse un *grid* en la red neuronal, hay que acceder a disco para cargar el respectivo *grid*. De hecho, los *grids* no se procesarán de uno en uno, sino que esto vendrá determinado por el tamaño del *mini-batch*.

2.2.4 Cuarta iteración

En esta iteración se pretende resolver el único problema existente en las fases previas al entrenamiento y pruebas de la red, consistente en la minimización del número de ficheros generados, así como de la cantidad de lecturas de disco necesarias.

2.2.4.1 Minimización del número de ficheros generados

Como ya se ha comentado en la sección 2.2.3.3, es necesario reducir el número de ficheros generados durante el preprocesamiento de datos dado que es difícilmente manejable para el sistema operativo y ralentiza el entrenamiento.

Dado que el ordenador en el que se va a realizar la experimentación cuenta con un tamaño de memoria principal de 64GBytes, que es más que suficiente para cargar todo el *dataset* de entrenamiento, se decide cargar el conjunto de entrenamiento completo en memoria principal y con esto quedaría resuelto el problema del acceso a disco.

El *framework* PyTorch proporciona una clase *Dataset* que permite implementar la funcionalidad para cargar un *dataset* customizado mediante herencia, siendo necesario implementar 3 métodos: el constructor *(__init__),* en el que se cargará el *dataset* completo en memoria y lo se guardará como atributo de clase; el método *__getitem__(item),* que será llamado desde un cargador de PyTorch (*DataLoader*), en forma de lotes (*mini-batches*), para servir como entrada a la red neuronal; y el método *__len__,* que proporciona la cantidad de ejemplos en el *dataset*.

Desde el método *__getitem__* se devolverá tanto el grid de vóxeles como un tensor de PyTorch de dimensiones $[C, N, N, N]$, donde C indica la cantidad de

canales de entrada (por ejemplo, en imágenes, los canales de entrada suelen ser 3: RGB; en este caso, $C = 1$), y N el número de vóxeles dentro del grid. Al almacenarse todo el dataset en memoria principal, únicamente habrá que devolver un tensor formado con la tupla especificada por el parámetro *ítem*, evitando la operación de acceso a disco (almacenamiento secundario).

Para conseguir que todos los *grids* de vóxeles permanezcan en la memoria principal y reducir por tanto el número de ficheros que se generan, es necesario generar un único fichero que pueda cargarse de forma completa. Se hace uso de la librería *json* de Python con el método *json.dumps(dict)* para conseguir una cadena de caracteres a partir del *grid* disperso que representa al grid. Esto se hará una vez por cada *grid* generado, y se almacenan todos en un mismo fichero *.csv* con el siguiente formato (véase Tabla 2.2):

Grid de vóxeles en formato JSON	Etiqueta de clase
{"1": {"7": {"9": 1}, "9": {"11": 1}, "11": {"13": 1}}, …, "15": {"13": 1}}}	4
…	…
M	$Etiqueta \in [0, N-1]$

Tabla 2.2: Estructura del fichero de entrenamiento en iteración 4

Sea M el número de ejemplos del conjunto de entrenamiento y sea N el número de clases del problema. Recordemos que **las etiquetas de clase deben reducirse del rango $[1, N]$ al rango $[0, N-1]$ para que el *framework* PyTorch pueda trabajar correctamente**.

De esta manera, se obtiene un único fichero de texto (tan voluminoso como la suma de todos los ficheros en la solución anterior), que es fácilmente tratable para el sistema operativo. Por tanto, con esta estrategia se optimiza el proceso de lectura del conjunto de entrenamiento desde disco, y se puede comenzar ensamblar una primera versión de la red neuronal.

Por otra parte, **el fichero *.csv*** que se utilizaba anteriormente para tener las referencias a los ficheros, y en el que quedaba registrada la etiqueta de cada *grid*, **queda obsoleto** al permanecer toda la información en el mismo fichero. El fichero con los *grids* de vóxeles para validación tiene la misma estructura que el fichero de entrenamiento.

2.2.4.2 Implementación de la red

Una vez implementada y depurada la parte del preprocesamiento de los datos y voxelización respectiva, se procede a implementar una primera versión de la red neuronal.

Como una primera versión, se implementa usando PyTorch una nueva clase *Net* que hereda de *nn.Module* y que representa a la red neuronal, siendo necesario implementar los métodos *__init__*, en el que se definen todas las capas necesarias como atributos de clase y que a su vez tiene que llamar al constructor de la superclase; y por otra parte, el método *forward(x)*, que recibe como argumento un ejemplo, y define cómo se hace la propagación hacia adelante.

Siguiendo el esquema propuesto en la sección 2.1.3, se añade una función de activación ReLU tras cada capa de convolución y tras las dos primeras capas *fully-connected*, con el objeto de que la red pueda aproximar cualquier función de naturaleza no lineal y así extraer patrones más complejos. En la última capa *fully-connected* se añade una función Softmax tal y como propone la arquitectura (Huang & You, 2016).

Tras una serie de pruebas, el modelo no conseguía ajustarse con la función Softmax tras la última capa, por lo tanto, se decide descartarla. Todos los parámetros relativos a configuración de los experimentos, pruebas, y resultados se indicarán en la sección 3.

Suprimiendo la función Softmax, se lanza una primera experimentación con los parámetros especificados en 3.1.1, 3.1.2 y el modelo se sobreajusta. Esto es, la red neuronal sobreaprende los datos de entrenamiento y es incapaz de hacer

predicciones correctas sobre datos que no se han utilizado en el entrenamiento, como validación o test.

2.2.4.3 Conclusiones

Sobre el sistema implementado, el paso siguiente consiste en realizar diferentes experimentos y determinar cuál es la configuración de la red con la que se obtienen mejores resultados.

Tras los primeros experimentos lanzados, se observa que la red sufre de sobreajuste. Por tanto, en la siguiente iteración el objetivo será una solución para el mismo.

2.2.5 Quinta iteración

En esta quinta iteración, el sistema está implementado en su totalidad, y la única tarea pendiente consiste en lanzar una serie de experimentos y realizar modificaciones sobre los parámetros del sistema (incluyendo la red neuronal) para conseguir la precisión y ajuste óptimos.

Como ya se ha comentado, todos los parámetros relevantes a la experimentación se expondrán en la sección 3. Por tanto, en esta iteración se describirán qué alternativas se han considerado para tratar el sobreajuste y realizar la experimentación.

2.2.5.1 Capas de *Dropout*

Una capa de *Dropout* consiste en un operador probabilístico que se posiciona a la salida de una capa de la red, y la única misión que tiene es anular, mediante cierta probabilidad definida a priori, algunas de las salidas de las neuronas reduciendo estos valores a 0 e impidiendo que el valor calculado se propague. Está demostrado matemáticamente que con esta acción se consigue resolver el problema del sobreajuste (con una correcta selección de la probabilidad para anular una salida).

Así, nuestra propuesta para solucionar el sobreajuste parte de aplicar capas de *Dropout* tanto tras las capas *fully-connected* como tras las capas de convolución.

Se realizará en diferentes configuraciones para comprobar la efectividad de las mismas.

2.2.5.2 Incremento del tamaño del *grid*

Otro de los aspectos a tratar durante la experimentación es considerar diferentes tamaños para el *grid* de vóxeles. A mayor número, mayor vecindario y mayor es el nivel de información de cada punto con respecto a sus vecinos.

Así, dado que los métodos para generación de *grids* de vóxeles están parametrizados, es posible variar la configuración de una manera rápida y sencilla.

2.2.5.3 Conclusiones

Llegado hasta este punto, la implementación de software ha sido completada y es el momento de comenzar a realizar experimentos de más alto nivel y envergadura, con el objeto de verificar la calidad de la solución propuesta.

3 EXPERIMENTACIÓN, RESULTADOS Y DISCUSIÓN

En esta sección se pretende abordar una experimentación detallada sobre el sistema propuesto. Se ha realizado un total de 9 experimentos, variando los diferentes parámetros e hiperparámetros.

El objetivo es detallar las peculiaridades de cada experimento, y finalmente recoger a modo de conclusión qué configuración es la óptima para nuestro sistema de entre las contempladas.

3.1 Experimentaciones y pruebas

A continuación, se detallan los 9 experimentos lanzados sobre la arquitectura propuesta para el sistema. El primer paso antes de realizar cualquier experimento consiste el realizar el **muestreo** de las 15 nubes etiquetadas para entrenamiento con una precisión $P = 0{,}2\,m$, seguido de la voxelización, con un tamaño de vóxel $t = 0{,}2\,m$, con las técnicas de aumento de datos pertinentes, así como distribución de ejemplos en conjuntos de entrenamiento y desarrollo (validación).

Como especificación general, en los experimentos 1 a 5 (inclusive) se ha utilizado un tamaño para el *grid* de $\mathbf{20^3}$ **vóxeles**, y una proporción de **90% entrenamiento y 10% validación** para los conjuntos. En los experimentos 6 a 9 (inclusive) se ha utilizado un tamaño de $\mathbf{28^3}$ **vóxeles** y una proporción de **80% entrenamiento y 20% validación**.

La red utiliza un tamaño de *kernel* de 5^3 en las capas convolucionales, con un *stride* de 1. En las capas de *pooling* se utiliza un tamaño de *kernel* de 2^3 y un *stride* de 2. En el caso de la capa de entrada de la red *fully-connected*, tendrá un tamaño de 160 unidades en el caso de *grids* de 20^3 vóxeles, y de 1280 unidades en el caso de grids de 28^3 vóxeles. La capa oculta estará fijada a 300 unidades ocultas, mientras que la capa de salida tendrá 8 unidades, una por cada clase distinguible en el problema.

Los parámetros especificados arriba son constantes para todos los experimentos. En todos ellos se ha utilizado un tamaño de *batch* de 32 ejemplos, lo

que genera un tensor de entrada para la red de $[32, 1, N, N, N]$ donde N es el número de vóxeles en una dimensión del *grid*, así como una **disminución del ratio de aprendizaje del 95%** en cada *epoch*.

Como algoritmo de optimización se ha utilizado el **Gradiente Descendente Estocástico** con $momentum = 0{,}9$, mientras que como función de coste se ha optado por **Entropía Cruzada**, especial para problemas de clasificación.

Leyenda para las tablas de parámetros

- **E**: número de *epochs*.
- **LR**: ratio de aprendizaje
- **FConv1**: número de filtros en la primera capa de convolución.
- **FConv2**: número de filtros en la segunda capa de convolución.
- **D**: aplicación de capas de *Dropout* tras las dos primeras capas *fully-connected.*
- **D3D**: aplicación de capas de *Dropout* 3D tras ambas capas de convolución 3D.

3.1.1 Experimento 1

3.1.1.1 Hiperparámetros

E	LR	FConv1	FConv2	D	D3D
30	0,001	20	20	No	No

Tabla 3.1: Hiperparámetros del experimento 1

Como primera aproximación para probar el enfoque propuesto, se escoge una configuración de hiperparámetros sencilla. Esta configuración es propuesta en 2.2.4.

Se el experimento y tarda un total de 4 horas en completar los 30 epochs.

3.1.1.2 Medida del ajuste

En las siguientes figuras se observa una medida del ajuste de la red en este primer entrenamiento.

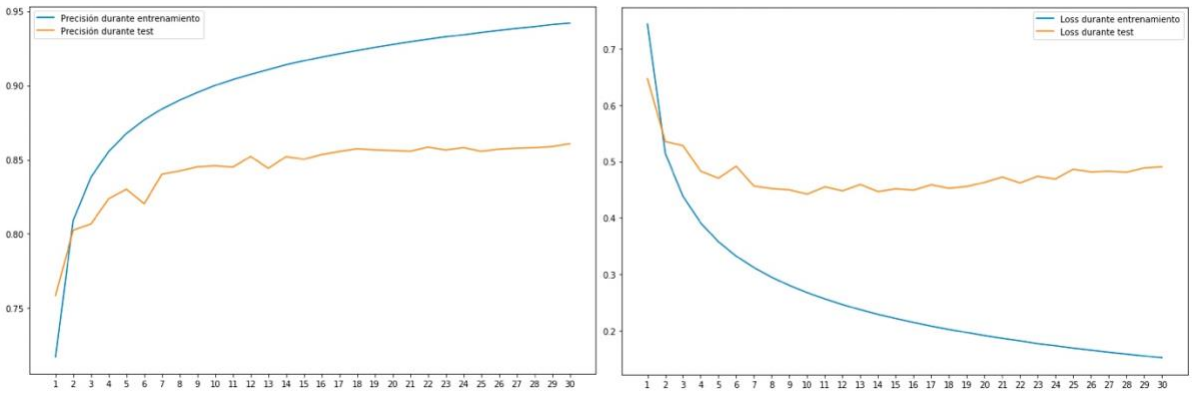

Figura 3.1: Ajuste de la red en experimento 1

La red consigue ajustarse a los datos de entrenamiento dando una precisión de casi un 95%. En cambio, este valor no resulta de interés, pues es apreciable cómo en el test (validación) tan solo consigue un 85%.

En la gráfica de coste (*loss*) se observa cómo existe un punto (a partir del 6º *epoch*) en el que una gráfica comienza separarse de la otra más de lo habitual.

Ambas gráficas recogidas arriba muestran un claro signo de **sobreajuste**, esto es, el modelo entrenado consigue clasificar muy bien los datos con los que ha sido entrenado, pero empieza a comportarse peor con los datos de test a medida que el entrenamiento avanza, **sobreajustando** la distribución de los datos de entrenamiento. Es **necesaria una estrategia para evitar este sobreajuste**.

Tras 30 *epochs*, el modelo consigue dar una precisión del 94,203% en el **entrenamiento** y del 86,081% en la **validación**.

3.1.1.3 Matriz de confusión

Para ahorrar espacio en los encabezamientos de la matriz de confusión, se identifica cada clase según la siguiente leyenda en todos los experimentos:

- **A**: Terreno artificial.
- **B**: Terreno natural.

- **C**: Vegetación alta.
- **D**: Vegetación baja.
- **E**: Edificios.
- **F**: Paisaje accidentado.
- **G**: Artefactos de escaneo.
- **H**: Coches.

		Etiquetas predichas							
		A	B	C	D	E	F	G	H
Etiquetas reales	A	1906	263	0	10	10	34	13	10
	B	322	1592	3	34	14	61	8	10
	C	0	6	2020	89	45	43	19	1
	D	18	37	131	1826	62	110	16	16
	E	9	23	17	46	1924	144	48	12
	F	27	37	27	93	150	1731	111	32
	G	15	1	6	8	43	80	2046	18
	H	15	10	0	17	10	37	27	2095

Tabla 3.2: Matriz de confusión para el experimento 1

Los elementos en la **diagonal principal** corresponden a los ejemplos que han sido correctamente clasificados.

3.1.2 Experimento 2

3.1.2.1 Hiperparámetros

E	LR	FConv1	FConv2	D	D3D
30	0,01	20	20	No	No

Tabla 3.3: Hiperparámetros del experimento 2

En este segundo experimento se pretende probar la influencia del ratio de aprendizaje en el entrenamiento para este conjunto de datos, con lo cual, se aumenta en un factor de 10 veces mayor.

A priori se espera que este experimento genere peores resultados que el anterior pues el ratio de aprendizaje es demasiado grande, por ello es necesario señalar que se trata de una prueba y que el objetivo en este caso no es mejorar lo anterior.

3.1.2.2 Medida del ajuste

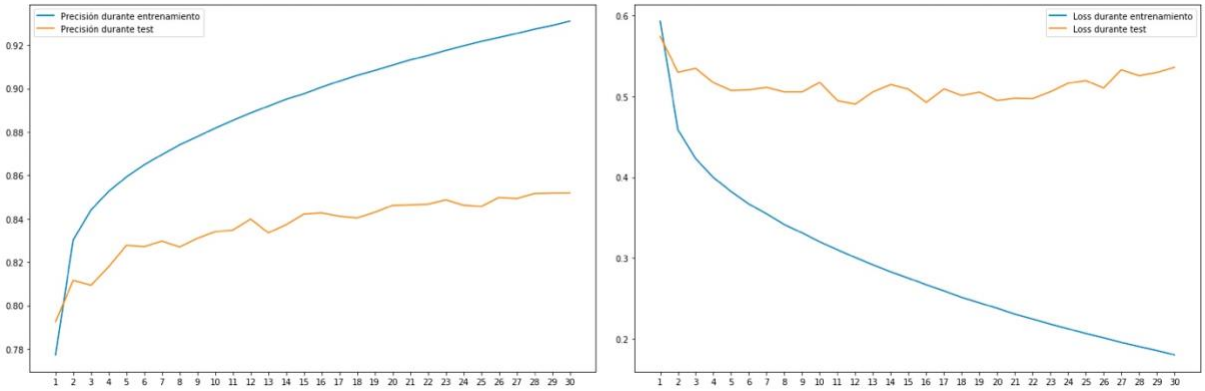

Figura 3.2: Ajuste de la red en experimento 2

El incremento tan severo en el ratio de aprendizaje no mejora el sobreajuste, y de hecho empeora el ajuste. Al dar saltos poco precisos durante la búsqueda de la configuración óptima para los parámetros de la red, es muy difícil encontrar la mejor solución.

Normalmente, el algoritmo de entrenamiento se estanca dando saltos en torno a la solución óptima, esto es, escoge configuraciones cercanas a la misma sin llegar a alcanzarla nunca.

Se puede ver cómo desde el 4º *epoch*, la función de coste del test comienza divergir del entrenamiento, siendo claro signo de sobreaprendizaje. Es necesario encontrar una buena alternativa para solucionar el sobreajuste.

La precisión tras 30 *epochs* es del 93,111% en el **entrenamiento** y del 85,183% en la **validación**.

3.1.2.3 Matriz de confusión

		Etiquetas predichas							
		A	B	C	D	E	F	G	H
Etiquetas reales	A	1878	300	0	13	6	33	10	6
	B	326	1582	5	33	20	52	15	11
	C	0	1	2032	100	33	38	15	4
	D	24	61	156	1788	47	100	16	24
	E	10	19	30	46	1917	155	36	10
	F	33	41	36	94	175	1713	89	27
	G	18	5	7	12	62	104	1990	19
	H	10	13	2	19	11	49	25	2082

Tabla 3.4: Matriz de confusión para el experimento 2

3.1.3 Experimento 3

3.1.3.1 Hiperparámetros

E	LR	FConv1	FConv2	D	D3D
60	0,001	20	20	Sí	No

Tabla 3.5: Hiperparámetros del experimento 3

En este experimento, el número de *epochs* se ha incrementado a 60, con el objetivo de profundizar sobre el resultado del primero; y también se han añadido capas de *Dropout* tras las dos primeras capas *fully-connected* de la red, con el objeto de solucionar el sobreajuste.

3.1.3.2 Medida del ajuste

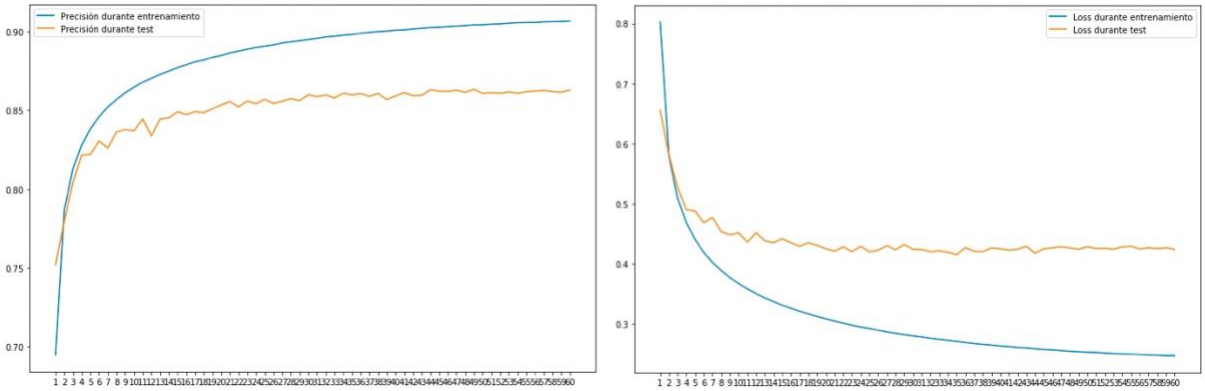

Figura 3.3: Ajuste de la red en experimento 3

El hecho de incorporar las capas de *Dropout* hace que la precisión durante el entrenamiento no alcance el 95% obtenido durante el primer entrenamiento aun cuando el número de *epochs* es del doble.

Sin embargo, es apreciable una menor divergencia entre ambas gráficas tanto en funciones de coste como en funciones de precisión, lo que implica una reducción importante en el sobreajuste de la red.

Tras 60 *epochs* de entrenamiento, se consigue una precisión del 90,663% en el **entrenamiento** y una precisión del 86,286% en la **validación**.

3.1.3.3 Matriz de confusión

		Etiquetas predichas							
		A	B	C	D	E	F	G	H
Etiquetas reales	A	1863	318	0	14	9	21	11	10
	B	275	1649	2	36	18	39	14	11
	C	0	1	2067	84	28	22	20	1
	D	12	56	135	1834	47	103	10	19
	E	9	19	21	56	1914	157	38	9
	F	28	50	33	107	125	1724	102	39
	G	16	4	8	17	46	88	2022	16
	H	6	8	0	23	8	39	24	2103

Tabla 3.6: Matriz de confusión para el experimento 3

3.1.4 Experimento 4

3.1.4.1 Hiperparámetros

E	LR	FConv1	FConv2	D	D3D
60	0,001	20	20	Sí	Sí

Tabla 3.7: Hiperparámetros del experimento 4

Para continuar con la disminución del sobreaprendizaje, se toma la decisión aplicar capas de *Dropout* tras las capas de convolución (*Dropout 3D*). El objetivo es que la red se vea tan perjudicada como sea posible durante el entrenamiento, y que no extraiga ningún patrón particular de los datos de entrenamiento, de manera que posteriormente se comporte mejor en la validación y por tanto el conocimiento extraído sea de mayor calidad.

3.1.4.2 Medida del ajuste

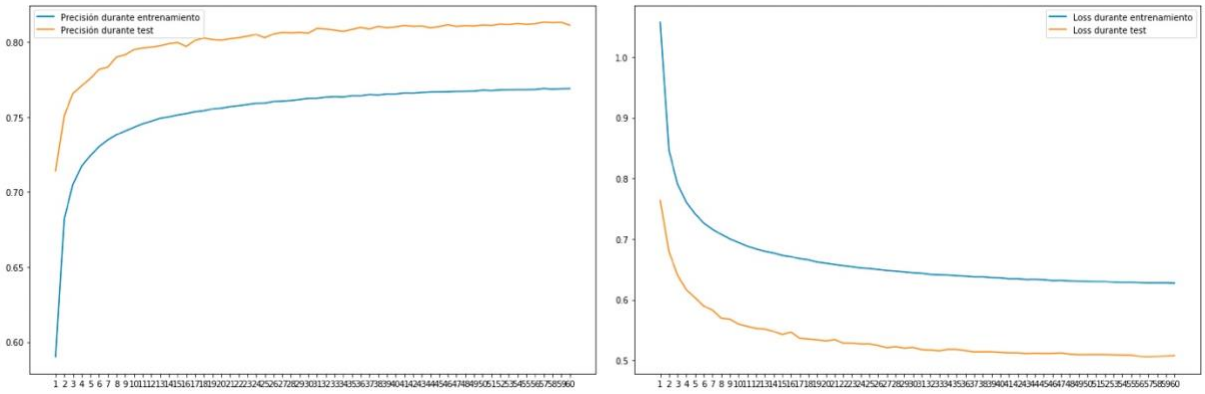

Figura 3.4: Ajuste de la red en experimento 4

La red consigue ajustarse con un porcentaje de precisión más bajo en el entrenamiento que en la validación.

En virtud de lo acontecido durante el entrenamiento, la red consigue una precisión mayor durante la validación, ya que no ha sobreaprendido la distribución de los datos de entrenamiento. Sin embargo, esta apuesta por evitar el sobreaprendizaje provoca un perjuicio sobre la precisión a nivel general, obteniendo un modelo no sobreajustado, pero mal entrenado.

Tras 60 *epochs* de entrenamiento, se consigue una precisión del 76,888% en el **entrenamiento** y una precisión del 81,123% en la **validación**.

3.1.4.3 Matriz de confusión

		Etiquetas predichas							
		A	B	C	D	E	F	G	H
Etiquetas reales	A	1892	316	0	13	0	6	10	9
	B	402	1501	3	54	19	32	16	17
	C	0	2	2039	116	9	36	20	1
	D	18	99	188	1736	40	86	16	33
	E	6	52	33	98	1820	150	54	10
	F	46	98	77	295	116	1274	204	98
	G	21	2	30	45	54	99	1935	31
	H	17	15	5	23	2	28	50	2071

Tabla 3.8: Matriz de confusión para el experimento 4

3.1.5 Experimento 5

3.1.5.1 Hiperparámetros

E	LR	FConv1	FConv2	D	D3D
60	0,001	10	20	Sí	No

Tabla 3.9: Hiperparámetros del experimento 5

El objetivo de este experimento es comprobar si se puede conseguir el mismo rendimiento reduciendo el número de filtros que se aplican en la primera capa de convolución. Con esto, se reduce el número de parámetros en esta capa a la mitad, y el entrenamiento se hace más rápido.

Otro factor a tener en cuenta es que esta capa extrae un menor número de características, lo cual resulta crucial para que la red pueda aprender sobre los datos de entrenamiento. Con este menor número de características, en la segunda capa de convolución se extraerán en consecuencia un menor número de características de más alto nivel. El número de filtros continúa siendo 20 en la segunda capa, pero

al ser menor en la primer, podría haber algún filtro que esté extrayendo la misma característica que otro filtro.

3.1.5.2 Medida del ajuste

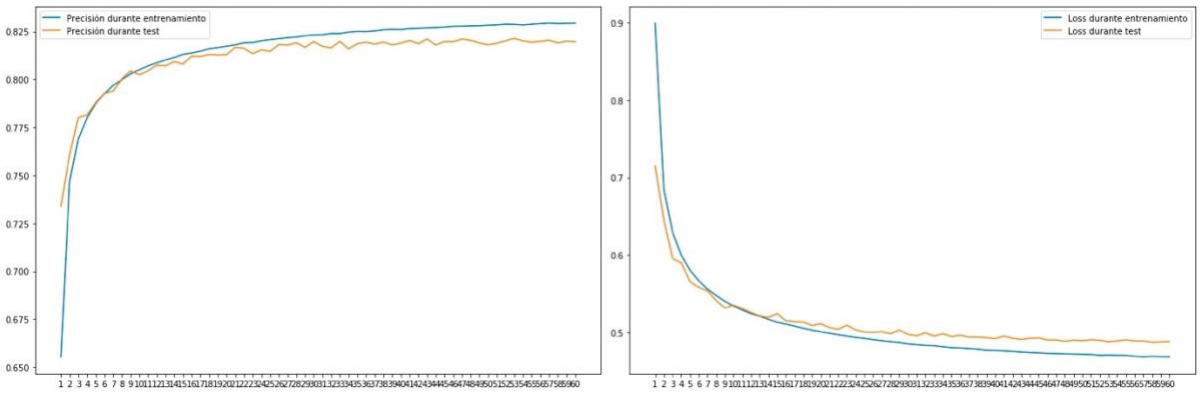

Figura 3.5: Ajuste de la red en experimento 5

Es apreciable que se ha resuelto el problema del sobreajuste en este experimento, pues ambas gráficas están muy poco separadas y no existe un punto de divergencia notable.

Por otra parte, el hecho de reducir el número de filtros en la primera capa de convolución empeora el entrenamiento, pues el porcentaje de acierto que se da en este experimento es menor que en 3.1.3, a favor de una solución para el sobreajuste de la red.

Como se puede observar, se da una primera aproximación hacia una configuración óptima para este enfoque, pues se han resuelto la mayoría de problemas descritos anteriormente.

Tras 60 *epochs* de entrenamiento, se consigue una precisión del 82,933% en el **entrenamiento** y una precisión del 81,959% en la **validación**.

3.1.5.3 Matriz de confusión

		Etiquetas predichas							
		A	B	C	D	E	F	G	H
Etiquetas reales	A	1836	346	0	13	7	21	10	13
	B	332	1543	3	48	31	50	14	23
	C	0	0	2008	122	25	41	24	3
	D	15	75	187	1652	96	124	17	50
	E	3	31	28	67	1868	169	45	12
	F	39	68	55	172	160	1485	151	78
	G	19	3	7	28	51	122	1949	38
	H	11	12	4	24	7	41	38	2074

Tabla 3.10: Matriz de confusión para el experimento 5

3.1.6 Experimento 6

3.1.6.1 Hiperparámetros

E	LR	FConv1	FConv2	D	D3D
60	0,01	20	20	Sí	Sí

Tabla 3.11: Hiperparámetros del experimento 6

A partir de este experimento se considera un tamaño para el *grid* de vóxeles de 28^3. El objetivo es probar la influencia de un entorno más grande para cada punto en cuanto a nivel informativo. A mayor tamaño del entorno, mayor cantidad de información.

Se aplica *Dropout* tanto en las capas *fully-connected* como en las capas de convolución 3D.

Además, se aumenta en un factor de 10 el ratio de aprendizaje con el objeto de que la red aprenda más rápido. Tal y como se detalla en la literatura, el ratio de aprendizaje deberá oscilar entre $[10^{-4}, 10^{-2}]$, y la red tendrá un mejor entrenamiento cuanto más pequeño sea. En cambio, a menor ratio, más lento será el aprendizaje.

3.1.6.2 Medida del ajuste

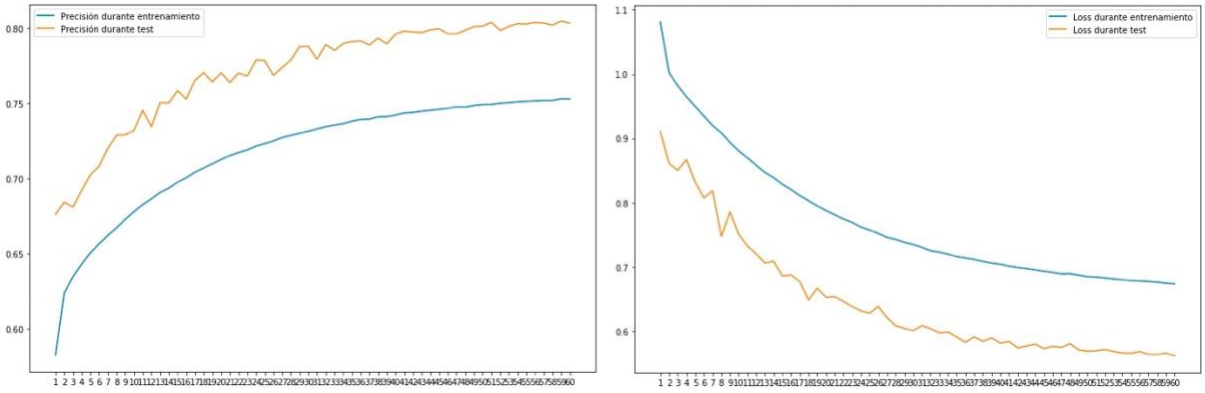

Figura 3.6: Ajuste de la red en experimento 6

Tal y como ocurrió en 3.1.4, el hecho de aplicar *Dropout* tras las capas de convolución hace que el entrenamiento se vea tan perjudicado como sea posible, de manera que la red no aprenda patrones concretos propios de los datos de entrenamiento y que solo estén presentes en algunos ejemplos. De esta forma, la red aprende a extraer los patrones más comunes y que están presentes en la mayoría de ejemplos.

Es por ello que consigue funcionar mejor durante la validación que durante el entrenamiento.

La función que representa la precisión durante la validación presenta una serie de picos (irregularidad) representando un entrenamiento no constante para la red. Esto se debe al elevado ratio de aprendizaje utilizado, esto es, durante el entrenamiento, el algoritmo de optimización (SGD) comenzará a dar saltos en torno a la función objetivo y cabe la posibilidad de moverse hacia estados peores, lo cuál se ve representado con una peor precisión y un coste de la función objetivo más alto.

Tras 60 *epochs* de entrenamiento, se consigue una precisión del 75,307% en el **entrenamiento** y una precisión del 80,353% en la **validación**.

3.1.6.3 Matriz de confusión

		Etiquetas predichas							
		A	B	C	D	E	F	G	H
Etiquetas reales	A	3738	633	0	42	13	27	16	16
	B	865	2940	1	139	26	81	3	23
	C	0	0	4116	215	46	46	10	4
	D	22	144	444	3439	98	185	15	69
	E	9	76	80	259	3688	284	29	23
	F	76	218	145	679	309	2437	399	162
	G	39	10	42	81	101	264	3829	57
	H	29	31	9	76	2	116	114	4042

Tabla 3.12: Matriz de confusión para el experimento 6

3.1.7 Experimento 7

3.1.7.1 Hiperparámetros

E	LR	FConv1	FConv2	D	D3D
60	0,001	20	20	Sí	No

Tabla 3.13: Hiperparámetros del experimento 7

En este experimento se pretende probar la calidad de la red cuando se trabaja en entornos más grandes, pues recordemos que a partir del experimento 3.1.6, los *grids* de vóxeles son de 28^3. Se planeta un experimento análogo a 3.1.3, pero con una variación de los datos de entrada.

3.1.7.2 Medida del ajuste

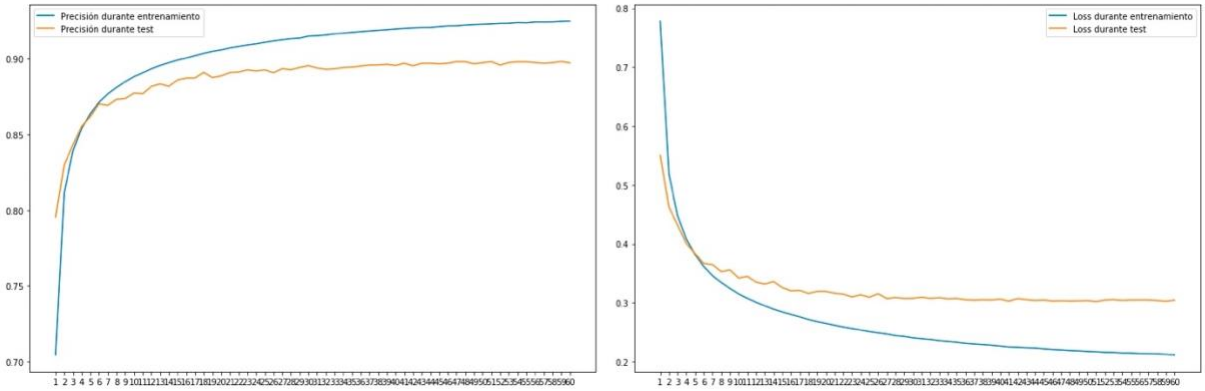

Figura 3.7: Ajuste de la red en experimento 7

Ambas gráficas tienen un aspecto muy similar al resultado del experimento 3.1.3. Por tanto, la duda es si un mayor entorno para cada punto consigue mejorar el porcentaje de acierto obtenido.

Tras 60 *epochs* de entrenamiento, se consigue una precisión del 92,494% en el **entrenamiento** y una precisión del 89,732% en la **validación**.

Por tanto, la respuesta a la pregunta es que **con un mayor entorno local para cada punto se consigue un mejor rendimiento tanto en el entrenamiento de la red como en la posterior validación**.

3.1.7.3 Matriz de confusión

		Etiquetas predichas							
		A	B	C	D	E	F	G	H
Etiquetas reales	A	3931	407	0	32	11	56	20	28
	B	462	3460	2	51	16	72	6	9
	C	0	3	4242	105	30	39	16	2
	D	21	71	195	3850	75	161	14	29
	E	19	31	36	57	4025	236	33	11
	F	60	89	65	195	179	3587	194	56
	G	33	6	10	19	48	127	4154	26
	H	18	10	5	14	8	39	50	4275

Tabla 3.14: Matriz de confusión para el experimento 7

3.1.8 Experimento 8

3.1.8.1 Hiperparámetros

E	LR	FConv1	FConv2	D	D3D
60	0,0001	20	20	Sí	No

Tabla 3.15: Hiperparámetros del experimento 8

En línea de los resultados obtenidos en el experimento inmediato anterior, se plantea un nuevo experimento reduciendo en un factor de 10 el ratio de aprendizaje utilizado.

El objetivo de esta reducción es intentar estabilizar el aprendizaje, de forma que la función objetivo tenga un incremento lo más suave posible en precisión, y una disminución equiparable en cuanto a función de coste.

3.1.8.2 Medida del ajuste

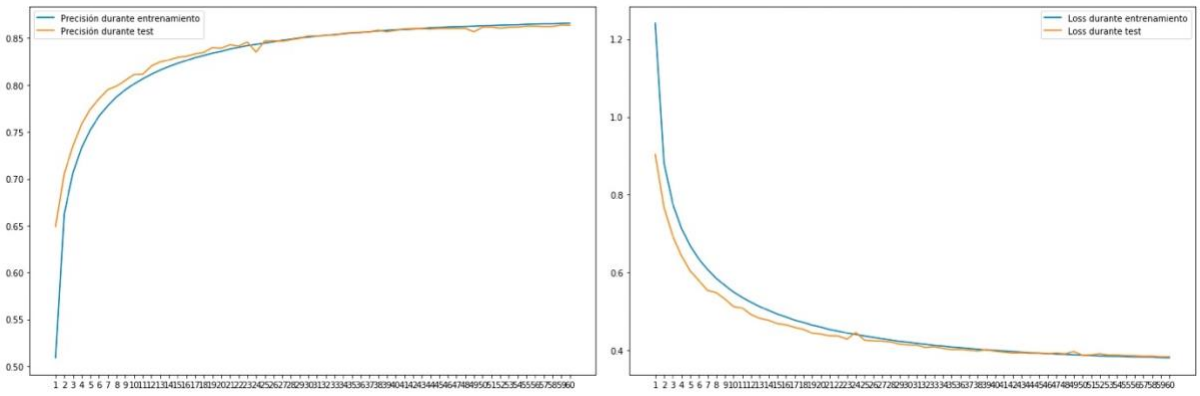

Figura 3.8: Ajuste de la red en experimento 8

Hasta el momento, se trata del mejor resultado obtenido de entre todos los experimentos realizados. La disminución del ratio de aprendizaje induce estabilidad en ambas funciones; los *grids* de mayor tamaño añaden información al aprendizaje y posterior validación, mientras que las capas de *Dropout* tras las capas *fully-connected* se comportan correctamente impidiendo el sobreajuste.

A medida que el entrenamiento se acerca a su fin (60 *epochs*) se hace prácticamente imposible encontrar una diferencia entra ambas gráficas tanto en precisión como en la función de coste.

Tras 60 *epochs* de entrenamiento, se consigue una precisión del 86,601% en el **entrenamiento** y una precisión del 86,376% en la **validación**.

Así, a menos que el último experimento mejore el estado actual, será este experimento el que nos servirá para elaborar una conclusión sobre el método propuesto, ya que la configuración escogida para el mismo es la óptima, y por tanto es el más representativo de entre todos los realizados.

3.1.8.3 Matriz de confusión

		Etiquetas predichas							
		A	B	C	D	E	F	G	H
Etiquetas reales	A	3715	626	0	38	14	50	18	24
	B	547	3324	1	79	13	88	8	18
	C	0	4	4133	165	61	55	18	1
	D	17	94	168	3728	88	269	15	37
	E	11	48	42	109	3903	285	37	13
	F	56	155	58	310	235	3331	204	76
	G	41	2	9	21	58	242	4008	42
	H	20	16	2	27	3	94	54	4203

Tabla 3.16: Matriz de confusión para el experimento 8

Dado que se trata del experimento con el mejor resultado, se toma la decisión de mostrar una matriz de confusión con los porcentajes de cada ejemplo. Los elementos sumados por filas representan al número total de ejemplos de cada clase. El elemento (i, j) de la Tabla 3.17 representa en tanto por ciento la porción de ejemplos de la clase i, etiquetados con la clase j.

		Etiquetas predichas							
		A	B	C	D	E	F	G	H
Etiquetas reales	A	82,8%	14,0%	0,0%	0,8%	0,3%	1,1%	0,4%	0,5%
	B	13,4%	81,5%	0,0%	1,9%	0,3%	2,2%	0,2%	0,4%
	C	0,0%	0,0%	93,1%	3,7%	1,4%	1,2%	0,4%	0,0%
	D	0,4%	2,1%	3,8%	84,4%	2,0%	6,1%	0,3%	0,8%
	E	0,2%	1,1%	0,9%	2,5%	87,7%	6,4%	0,8%	0,3%
	F	1,3%	3,5%	1,3%	7,0%	5,3%	75,3%	4,6%	1,7%
	G	0,9%	0,0%	0,2%	0,5%	1,3%	5,5%	90,6%	0,9%
	H	0,5%	0,4%	0,0%	0,6%	0,1%	2,1%	1,2%	95,1%

Tabla 3.17: Matriz de confusión para la precisión en el experimento 8

La precisión del modelo se ve gravemente perjudicada al etiquetar ejemplos de la clase F (Paisaje Accidentado), mientras que en el resto de clases llega a conseguirse una precisión del 95,1% en el mejor de los casos (Clase H: Coches). Resto de conclusiones en 3.2.

3.1.9 Experimento 9

3.1.9.1 Hiperparámetros

E	LR	FConv1	FConv2	D	D3D
60	0,001	10	10	Sí	Sí

Tabla 3.18: Hiperparámetros del experimento 9

La teoría a estudiar en este experimento, aunque redundante pues ya se ha encontrado una configuración óptima, consiste en estudiar si es posible conseguir un rendimiento equiparable aumentando el ratio de aprendizaje para que entrenamiento sea más rápido, reduciendo el número de filtros tanto en la primera capa de convolución como en la segunda, así como la aplicación de capas de *Dropout* para evitar el sobreajuste.

Este experimento trata, a grandes rasgos, de hibridar cada una de las técnicas aplicadas en los experimentos anteriores, con el objeto de probar si esta amalgama de técnicas consigue mejorar los resultados actuales.

3.1.9.2 Medida del ajuste

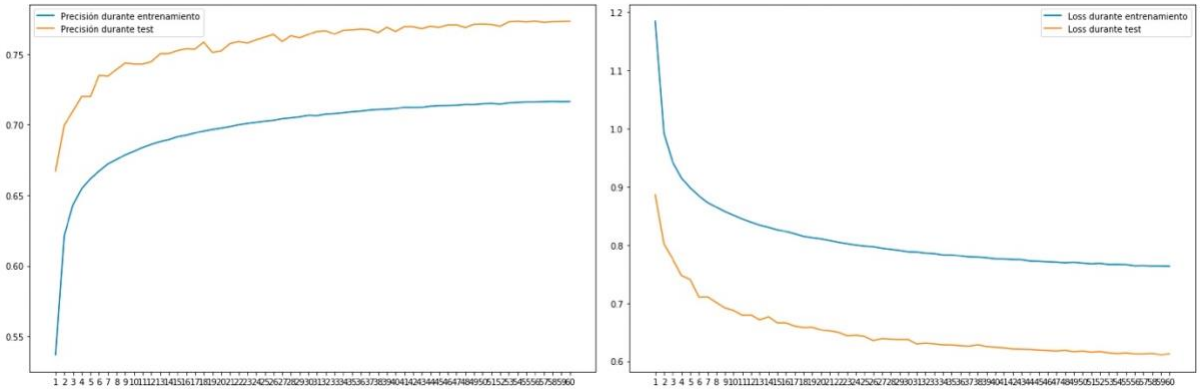

Figura 3.9: Ajuste de la red en experimento 9

El entrenamiento se mantiene estable y ninguna de las dos funciones presenta una irregularidad que sea símbolo de sobreajuste o ratio de aprendizaje demasiado alto.

La reducción del número de filtros reduce la cantidad de parámetros de la red, y acelera ligeramente el entrenamiento. En cambio, como consecuencia no se consigue un rendimiento tan bueno como en el entrenamiento anterior.

3.1.9.3 Matriz de confusión

		Etiquetas predichas							
		A	B	C	D	E	F	G	H
Etiquetas reales	A	3644	720	0	33	15	32	14	27
	B	988	2801	0	127	45	77	12	28
	C	0	2	3982	292	96	41	17	7
	D	25	201	267	3273	181	224	20	225
	E	6	118	43	217	3749	241	36	38
	F	69	265	112	683	373	2161	496	266
	G	40	21	53	90	101	303	3629	186
	H	27	39	11	163	14	126	110	3929

Tabla 3.19: Matriz de confusión para el experimento 9

3.2 Resultados y discusión

Tras haber completado los 9 experimentos anteriores, y habiendo analizado con detalle el resultado de cada uno de ellos, se puede concluir con que la mejor configuración y el mejor rendimiento quedan expuestos en el experimento 3.1.8.

Por tanto, las estrategias que han permitido conseguir tal configuración óptima han sido:

- **Aumentar el tamaño del *grid* de 20^3 a 28^3.** Con esto, el nivel de información que acompaña a cada punto es más representativo, mejorando la tarea de segmentación de la red.
- **Disminuir el ratio de aprendizaje hasta 0,0001.** En abundancia de datos, es posible la aplicación de un ratio de aprendizaje bajo, aunque esto suponga un aumento significativo del tiempo de entrenamiento. De esta forma, y con el número de *epochs* adecuado, se consigue que el desplazamiento por el espacio de soluciones sea lo más lento posible, y la solución óptima pueda ser alcanzada sin dar saltos en torno a ella.
- **Aplicar capas de *Dropout* tras las dos primeras capas *fully-connected.*** El operador de *Dropout* reduce el sobreajuste de la red anulando la salida de ciertas neuronas con una probabilidad preestablecida. El objetivo de esto es que la red no se enfoque en aprender patrones particulares que estén presentes solo en algunos ejemplos del conjunto de entrenamiento, sino que extraiga aquellos que son más comunes a todos los ejemplos del conjunto, de manera que se capture solo la señal de los datos y no también el ruido, como ya se explicó en 1.3.1.

Seleccionado el experimento que proporciona el rendimiento más óptimo (3.1.8), el siguiente paso consiste en tomar una nube de puntos como entrada, con los puntos sin etiquetar, y realizar la segmentación utilizando la red neuronal entrenada durante este experimento. La Figura 3.10 muestra el resultado de la nube etiquetada tras ser procesada por la red neuronal entrenada durante el experimento 3.1.8.

Señalar que el proceso de generación de *grids* durante el test difiere ligeramente del entrenamiento (2.2.1.2).

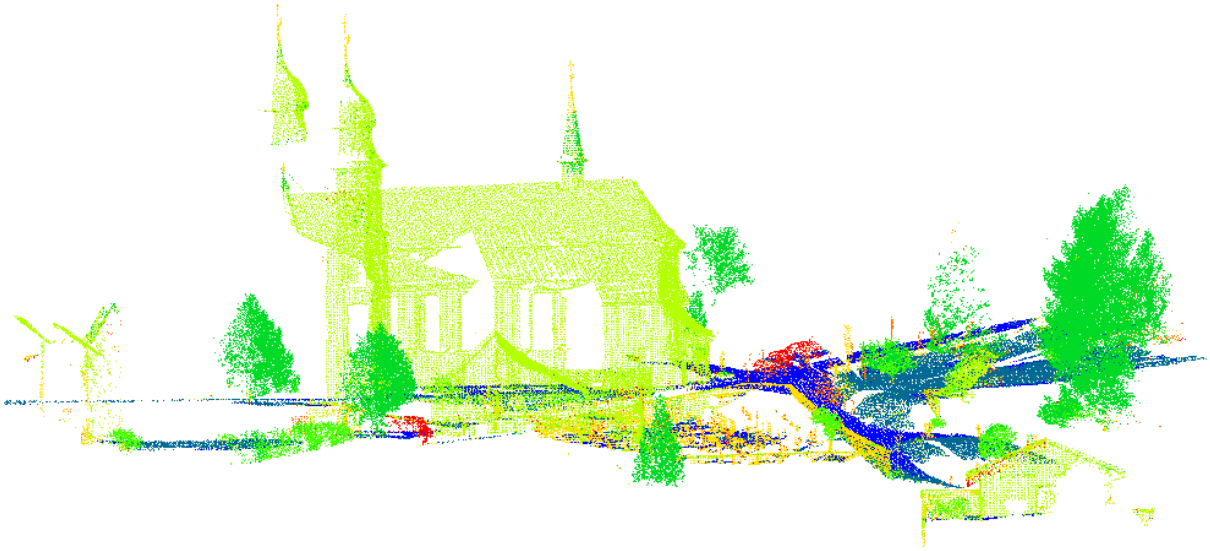

Figura 3.10: Nube de puntos del dataset Semantic 3D segmentada utilizando la red neuronal entrenada durante el experimento 8

A continuación, se presentan dos imágenes de la misma nube no etiquetada: la **primera** (Figura 3.11) corresponde a la segmentación que propone la red tras el primer *epoch* de entrenamiento, mientras que la **segunda** (Figura 3.12) es el producto de segmentar una vez finalizado el entrenamiento. Los puntos en color verde representan una asignación correcta de etiqueta, mientras que los puntos en color azul representan una asignación errónea.

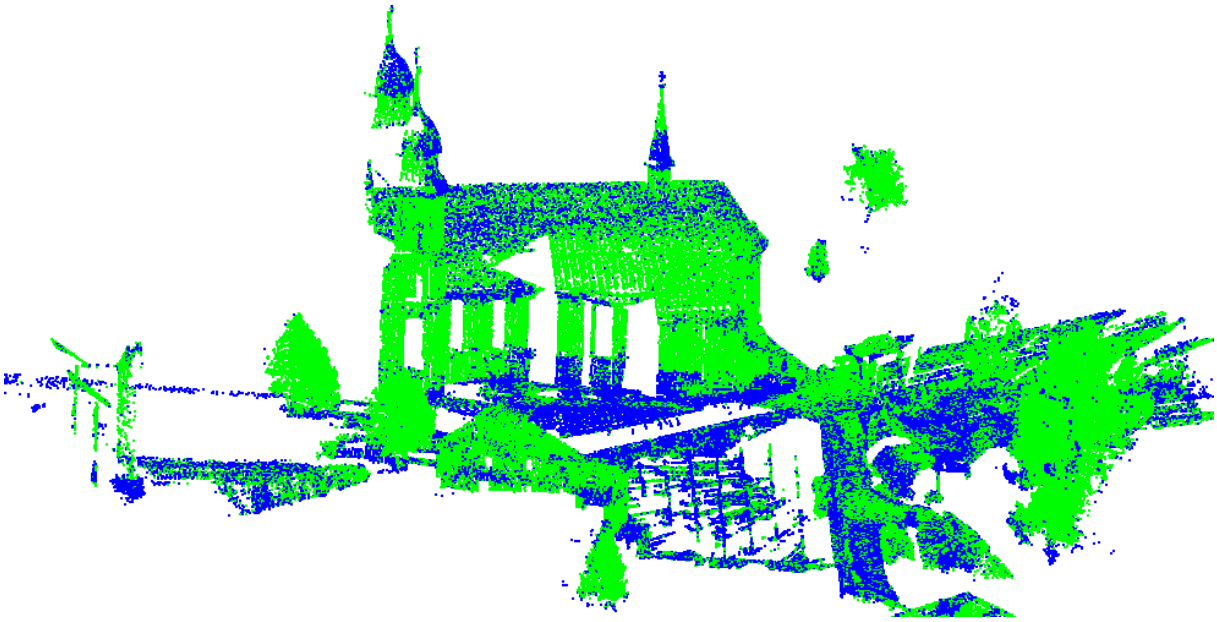

Figura 3.11: Etiquetado de la nube tras 1 epoch de entrenamiento

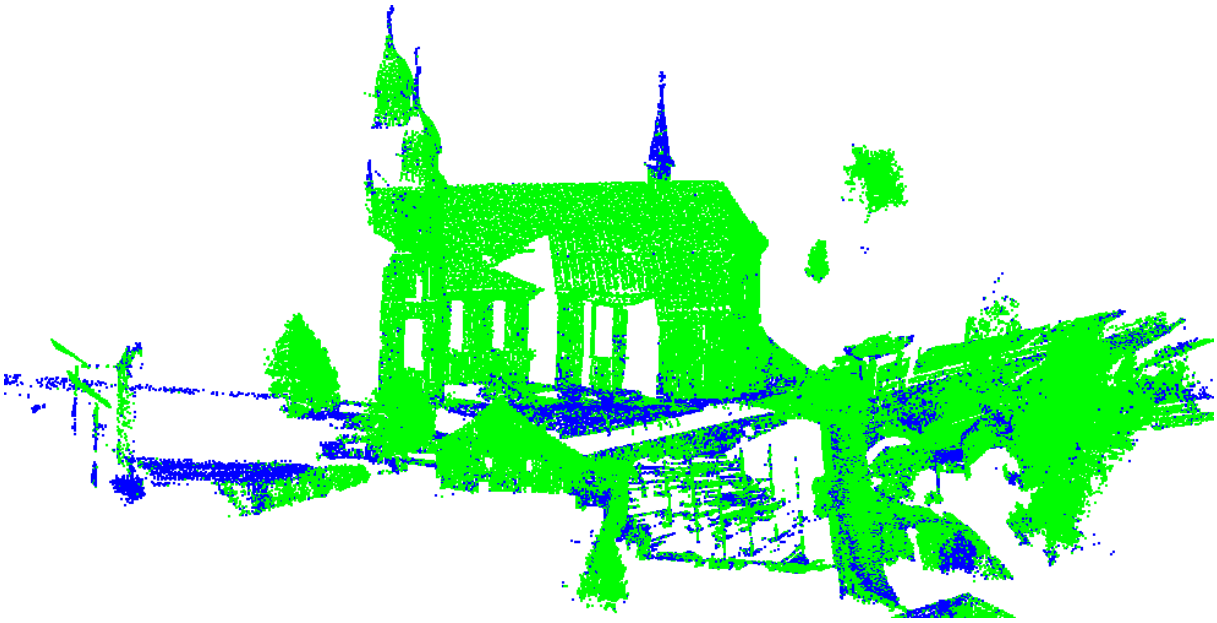

Figura 3.12: Etiquetado de la nube tras 60 epochs de entrenamiento

A partir de las matrices de confusión obtenidas, es posible determinar que la red consigue clasificar mejor las clases “vegetación alta” y “coches” en la mayoría de ocasiones. Para concluir esto, ha de observarse el valor que se encuentra en la **diagonal principal** para cada clase. Cuanto más alto este valor, significa que el

valor real concuerda con el valor predicho mayor número de veces (en mayor número de ejemplos del conjunto de validación).

Observando la matriz de confusión con los porcentajes de acierto, es posible comprobar que la clase más desfavorecida es la clase “Paisaje Accidentado”. El motivo es la elección de un tamaño de vóxel elevado; dado que los puntos que se integran en esta clase pueden ser de cualquier tipo que no sean clasificables en ninguna otra, esto es: tendido eléctrico, farolas, etcétera, algunos objetos podrían ser más pequeños que el tamaño de vóxel, y esto conlleva una pérdida de precisión que empeora el etiquetado.

4 CONCLUSIONES Y TRABAJOS FUTUROS

La **segmentación semántica en nubes de puntos 3D** forma parte de un área de investigación que irá cobrando mayor interés durante los próximos años. La necesidad de analizar la información espacial en arquitectura e ingeniería civil, el BIM, la gestión inteligente del territorio, y un largo etcétera de casos de uso. Por otra parte, la llegada del coche autónomo se potencia con esta tarea: el coche podría incorporar un sensor LiDAR y utilizar este dispositivo como medio de visión artificial, por lo que es necesario diseñar algoritmos que sean capaces de segmentar nubes de puntos en tiempo real.

En este Trabajo Fin de Grado se realiza una **revisión teórica de los principales métodos para segmentar en nubes de puntos utilizando técnicas de aprendizaje profundo**. De todas las técnicas estudiadas, **se escoge una de ellas para implementar, experimentar y concluir al respecto** sobre las peculiaridades del método.

Como conclusiones particulares, esta técnica es propensa a generar un gran volumen de información en formato de regiones voxelizadas a partir de una nube de puntos de entrada. El tratamiento de datos necesario consume recursos en tiempo y espacio, así como la voxelización genera un conjunto de datos de entrada muy voluminoso y difícilmente tratable de una manera eficiente. Por otra parte, el *modus operandi* de la técnica resulta sencillo, habiéndose expuesto que con una correcta elección de los hiperparámetros pueden conseguirse resultados muy **prometedores**.

En esta técnica **se ha utilizado únicamente la información espacial** $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ de cada punto, generando voxelizaciones de entornos locales que representan la **morfología** del espacio. Sin embargo, las nubes de puntos generadas mediante LiDAR contienen otros datos de interés en cada punto como el color RGB o la intensidad (medida de la reflectividad de un material cuando el láser del LiDAR impacta con la superficie de un objeto), y esta información podría añadirse al proceso para obtener una segmentación más precisa. La **intensidad**, por ejemplo, sería una medida de interés pues varía considerablemente de puntos que

pertenecen a una superficie de la categoría “edificio” con respecto a puntos que pertenecen a la categoría “vegetación alta” o “coche”.

Como **trabajos futuros**, podría estudiarse un método de segmentación semántica de nubes de puntos que se base en esta propuesta, y que también haga uso de los diferentes canales de información proporcionados por el sensor LiDAR como valor añadido a la propuesta actual. Esto es, no solamente podría representarse la morfología de la realidad, sino también una medida de la reflectividad de los objetos que la componen (usando la intensidad, por ejemplo). Además, la escasez de *datasets* para entrenamiento de algoritmos de aprendizaje automático, genera un problema cuando se intenta abordar una problemática de este tipo desde esta perspectiva. Por tanto, otro aspecto a tratar sería la generación de *datasets* de nubes de puntos etiquetadas entre las diferentes categorías semánticas del problema a tratar.

La tarea de **segmentación semántica de nubes de puntos constituye un área de interés en visión artificial** que cobrará especial relevancia en los próximos años motivada especialmente por la reciente llegada del coche autónomo. El incremento en potencia tanto en el *hardware* como en el *software* hace posible realizar un tratamiento automático e inteligente de estos densos volúmenes de datos, sumándose como un factor más al crecimiento del interés en la segmentación semántica.

5 APÉNDICES

5.1 Inventario del software implementado

Aunque se trata de un trabajo teórico-experimental, ha sido necesario implementar la solución propuesta para poder desarrollar los diferentes experimentos. Dado que el código fuente generado ha de ser un entregable, se toma la decisión de añadir un pequeño anexo describiendo los ficheros que componen el software entregado, así como los principales directorios necesarios para ejecutar el sistema. Dentro del directorio ***src*** adjunto constan los siguientes ficheros:

- ***config.py***. Rutas de los ficheros de datos así como algunos parámetros de configuración generales del sistema.
- ***dataset.py***. Implementación de la clase que utiliza internamente PyTorch para cargar el conjunto de datos que servirá para entrenamiento/validación/test. Se trata de una especialización de la clase *Dataset* de PyTorch donde es necesario implementar una serie de métodos como __len__ o __getitem__. Esta clase simplifica el trabajo en el momento de trabajar con un *dataset* customizado.
- ***files.py***. Implementación de funcionalidad útil para realizar el muestreo de las nubes o para cargar/almacenar ficheros desde disco.
- ***grid.py***. Clase GridDisperso. Esta clase sirve para generar un grid disperso dada una nube de puntos de entrada, de manera que se disponga de una estructura de datos eficiente en el momento de realizar consultas de vecinos para un punto en concreto. Además, esta clase alberga funcionalidad para realizar muestreos sobre la nube de puntos original. Dado que Python no contempla la existencia de varios constructores para una clase, es necesario seleccionar uno en concreto según la tarea a resolver. El código está documentado y se clarifica esto último.
- ***main.py.*** Fichero *main* del software. Desde este script se invoca al resto de métodos de la aplicación.

- ***model.py***. Definición de la clase que implementa una red neuronal. Esta clase especializa la clase *nn.Module*, y en el constructor se definen todas las capas de la red. Es necesario implementar el método *forward* qué define la manera en que se interconectan las diferentes capas y, por tanto, se realiza la propagación hacia adelante en la red. No es necesario establecer ningún otro parámetro para la red.
- ***train.py***. Implementación de código para realizar entrenamiento/validación/test con la red neuronal definida en la clase *Net* en el fichero *model.py*.
- ***voxelize.py***. Clase para realizar la voxelización dado un fichero de nubes de puntos como entrada. La especificación de cómo se hace la operación se describe tanto en secciones anteriores (véase 2.1.2) como en el propio código proporcionado.
- ***dataset_entrenamiento***. Directorio donde se almacenarán los ficheros con los *grids* de vóxeles generados.
- ***models***. En este directorio se almacenarán los modelos entrenados. Tras cada epoch de entrenamiento se almacenará en disco la configuración de pesos de la red en ese instante por si fuese necesario regresar a esa configuración en algún momento.
- ***nubes_puntos***. Directorio destino donde se guardarán las nubes muestreadas en formato *.txt*. Cada fichero corresponde a una nube, y en cada línea de cada fichero aparecerán los datos: *x y z label*, siendo respectivamente las coordenadas del punto y la etiqueta asociada.
- ***pointclouds/txt***. En este directorio permanecerán las nubes originales (sin muestrear). Dado que no se proporcionan junto al código fuente, será necesario remitirse al sitio web indicado en la documentación (véase 1.9.1) y descargar los ficheros originales.
- ***results***. Ficheros de resultados de entrenamientos, tanto ficheros de *log* con medidas de coste/ajuste en cada *epoch* para cada experimento como las matrices de confusión correspondientes (en formato de texto plano).

- ***orden.txt***. Este fichero se colocará en la raíz, e indica el orden que se seguirá cuando se esté realizando el muestreo de las nubes de puntos originales. En cada línea aparecerá un par de rutas relativas, una al fichero de puntos y otra al fichero de etiquetas para esos puntos, que estarán situados en el directorio ***pointclouds/txt***.

Para realizar pruebas con el código es necesario instalar Anaconda en el sistema operativo, crear un entorno virtual, e instalar todos los paquetes que se especifican en 1.10.

5.2 Publicaciones Científicas

A voxel-based deep learning approach for Point Cloud Semantic Segmentation. Congreso Español de Informática Gráfica (2019).

CEIG – Spanish Computer Graphics Conference (2019)
A. Jarabo and D. Casas (Editors)

A voxel-based deep learning approach for Point Cloud Semantic Segmentation

Miguel Díaz-Medina[1]*, José-Manuel Fuertes-García[1], Carlos-Javier Ogayar-Anguita[1], Manuel Lucena[1]

[1]Departamento de Informática. Universidad de Jaén

Abstract

Semantic segmentation has been a research topic in computer vision for decades. This task has become a crucial challenge nowadays due to emergence of new technologies such as autonomous driving. Nonetheless, most existing segmentation methods are not designed for handling the unstructured and irregular nature of 3D point clouds. We propose a voxel-based technique for point cloud data semantic segmentation of 3D point clouds using 3D convolutional neural networks. It uses local voxelizations for learning spatial patterns, and also corrects the imbalance of the data, something very problematic with 3D datasets.

CCS Concepts

•***Computing methodologies*** → *Computer Graphics; Machine Learning;*

1. Introducción

La segmentación de nubes de puntos es un área de gran interés en la que la mayoría de los métodos están todavía evolucionando. Existen numerosos ámbitos donde los avances en esta línea son imprescindibles, como la gestión inteligente del territorio, la planificación urbana, los vehículos autónomos, el modelado BIM o la arqueología, entre otros. En estos casos los datasets son de una gran magnitud, y los métodos clásicos, puramente geométricos, no trabajan correctamente o bien necesitan un tedioso procesamiento manual adicional. Por esta razón, es necesario automatizar el proceso en la medida de lo posible.

El objetivo de nuestra propuesta es un sistema de segmentación de nubes de puntos que las utiliza como entrada en su formato irregular y segmenta semánticamente los datos a partir de patrones aprendidos. Proponemos una técnica de regularización de la nube de puntos, que genera grids de vóxeles en diferentes entornos locales de la nube, espacio donde se asignará una etiqueta de clase. En base a este conjunto de datos voxelizados, se propone una arquitectura de red neuronal convolucional que contiene capas de convolución 3D que procesan los grids de entrada, extrayendo patrones en forma de características. De esta forma la red infiere la etiqueta de cada punto a partir de la morfología de su entorno local.

2. Trabajos previos

Cuando se decide aplicar aprendizaje profundo sobre modelos tridimensionales, existen varias alternativas para la representación de los datos: nubes de puntos, voxelizaciones, mallas de triángulos e imágenes RGB(D). Una red neuronal convolucional siempre debe usar datos de entrada del mismo tamaño. Al tratar imágenes, la resolución deberá ser fija (p. ej.: 1024×1024). En el caso de datos 3D, las representaciones basadas en imágenes o en vóxeles son las más adaptables, pero sufren de baja precisión y un uso excesivo de memoria en el caso de las voxelizaciones.

Existen métodos que proponen diseños de redes que aceptan conjuntos de puntos llevando a cabo un muestreo sobre la nube original, y trabajando siempre con el mismo número de elementos, lo que implica una pérdida importante de precisión. Destaca en este sentido PointNet++ [QYSG17], que está más enfocado a la clasificación de subconjuntos de puntos segmentados de la nube original. Por otra parte, existen otros métodos de clasificación que proponen la voxelización completa de objetos para entrenar una red neuronal convolucional 3D [MS15] [QSN*16]. Este enfoque también requiere modelos segmentados, a los que asignará una única etiqueta.

La mayoría de las propuestas actuales que tratan nubes de puntos mediante aprendizaje automático procesan modelos segmentados para poder realizar una clasificación. Este trabajo propone un algoritmo de segmentación semántica basado en técnicas de etiquetado de nubes de puntos [HY16] que permita realizar la tarea minimizando el conocimiento previo requerido.

3. Sistema propuesto

Nuestro enfoque se basa en la voxelización de entornos locales de una nube de puntos para entrenar una red neuronal que aprende patrones espaciales. Para ello, deben utilizarse muestras de tamaño

* Autor para correspondencia. E-mail: mdm00030@red.ujaen.es

6 DEFINICIONES Y ABREVIATURAS

6.1 Abreviaturas

- **BIM (Building Information Modeling).** Se trata de una metodología de trabajo colaborativa para la creación y gestión de un proyecto de construcción. Su objetivo es centralizar toda la información del proyecto en un modelo de información digital creado por todos sus agentes.
- **SGD: Stochastic Gradient Descent (Gradiente descendente estocástico).** Se trata de un algoritmo de optimización utilizado durante el entrenamiento de redes neuronales (back-propagation) para encontrar la configuración de parámetros óptima con la que el modelo resuelve mejor el problema.
- **SFM: Structure From Motion.** Algoritmo para la reconstrucción de modelos 3D a partir de imágenes 2D. Para ello, se buscan puntos en común en las diferentes imágenes, a partir del cual se reconstruye la escena 3D. Se trata de un procedimiento costoso computacionalmente.
- **LiDAR: Light Detection And Ranging.** Se trata de un dispositivo que permite determinar la distancia desde un emisor láser a un objeto o superficie utilizando un haz láser pulsado. La distancia al objeto se determina midiendo el tiempo de retraso entre la emisión del pulso y su detección a través de la señal reflejada. Se trata de uno de los principales medios para la obtención de nubes de puntos.
- **CAD/CAM: Computer-Aided Design/Computer-Aided Manufacturing.** Diseño asistido por computadora/Fabricación asistida por computador. Sus principales aplicaciones son en ingeniería y arquitectura, en programas como AutoCAD (AutoDesk).
- **HPC: High Performance Computing (Computacion de altas prestaciones).** Agregación de potencia de cálculo para resolver problemas complejos en ciencia, ingeniería o gestión. Para conseguir esto, la computación de altas prestaciones se apoya en tecnologías como *clusters*, supercomputadores o computación paralela.

- **GPU: Graphics Processor Unit (Unidad de Procesamiento Gráfico).** Se trata de la arquitectura de procesador específica dedicada al tratamiento de gráficos por computador. En los últimos años, estas unidades de procesamiento se han especializado en otras tareas de propósito general como la programación paralela. Se trata de un procesador SIMD.
- **SIMD: Simple Instruction Multiple Data.** Arquitectura de procesamiento consistente en múltiples núcleos de procesamiento que realizan la misma instrucción sobre conjuntos de datos diferentes. Por ejemplo, en *rendering*, existen los *shaders*, pequeños programas que se ejecutan en la GPU. Cada *shader* opera con un vértice en concreto o sobre un fragmento, pero todos realizan el mismo cálculo.
- **GRU: Gated Recurrent Unit.** Tipo de red neuronal recurrente.
- **LSTM: Long-Short Term Memory.** Tipo de red neuronal recurrente.
- **SIFT: Scale-Invariant Feature Transform.** Algoritmo para la reconstrucción de nubes de puntos utilizado en la técnica SFM.
- **SDK: Software Development Kit (Kit de desarrollo de software).**
- **IDE: Integrated Development Environment (Entorno integrado de desarrollo).**
- **CVS: Control Version System (Sistema de control de versiones).** Ejemplo: Git.
- **CUDA: Compute Unified Device Architecture (Arquitectura unificada de dispositivos de cómputo).** Tecnología propietaria de NVIDIA.
- **OpenGL: Open Graphics Library (Librería de gráficos).** Se trata de una API para trabajar con gráficos por computador mantenida por la compañía Khronos (https://www.khronos.org).
- **OpenCL: Open Computing Library (Librería de computación).** Estándar propuesto por Apple (https://www.apple.com) para realizar tareas de computación paralela, tanto en procesadores multinúcleo (SIMD) como en arquitecturas multiprocesador.

- **RMSE: Root Mean Square Error (Error cuadrático medio).** Función de coste para el entrenamiento de algoritmos de aprendizaje automático que suele utilizarse en tareas de regresión.
- **FBO: Framebuffer Object.** Extensión de OpenGL para realizar *rendering* offline, esto es, no visible en pantalla.
- **MBB: Minimal Bounding Box (Caja mínima envolvente).**
- **FPS: Furthest Point Sampling (Muestreo de punto lejano).** Algoritmo para la obtención de los puntos más lejanos entre sí, definidos dentro de un espacio métrico.

6.2 Definiciones

- **Característica.** Una imagen, tal y como se obtiene de los dispositivos de captura, carece de entidad semántica. En muchas ocasiones resulta conveniente extraer de ellas determinados atributos que faciliten su procesamiento. Estos atributos poseen mayor carga semántica que la colección de píxeles que componen la imagen: líneas, regiones, bordes, etcétera.
- **Filtro.** El término filtro proviene del dominio de la frecuencia, en el cuál se establecen procesos de filtrado que aceptan (dejan pasar) o rechazan ciertas componentes de la frecuencia. Cuando se trata de imágenes, un filtro se quedará únicamente con los bordes, o con la textura, en definitiva características de la imagen. Los filtros son la herramienta de las capas de convolución con las que se extraen las características.
- **Voxelizar.** Consiste obtener un modelo de vóxeles a partir de un modelo 3D en una representación diferente como podría ser una malla, una nube de puntos, o incluso una imagen RGB(D) con profundidad (D, Depth).
- **Hiperplano.** En geometria, un hiperplano es una extensión del concepto de plano. En un espacio unidimensional (como una recta), un hiperplano es un punto: divide una línea en dos líneas. En un espacio bidimensional (como el plano xy), un hiperplano es una recta: divide al plano en dos mitades. En un espacio tridimensional, un hiperplano es un plano

corriente: divide al espacio en dos mitades. Este concepto también puede ser aplicado a espacios de cuatro dimensiones o más, donde estos objetos divisores se llaman simplemente hiperplanos, ya que la finalidad de esta nomenclatura es la relacionar la geometría con el plano.

- ***Rendering*.** Se conoce como el proceso de generación de una imagen visible e inteligible para el ser humano, utilizando para ello un ordenador. El área que se ocupa del estudio del *rendering* es la informática gráfica.
- **Inteligencia Artificial.** Según la RAE (Real Academia Española), *"disciplina científica que se ocupa de crear programas informáticos que ejecutan operaciones comparables a las que realiza la mente humana, como el aprendizaje o razonamiento lógico"*. Se trata de un campo de las ciencias de la computación en auge y con un número de aplicaciones aún por conocer.
- **Vóxel.** Es una unidad cúbica que compone un objeto tridimensional. Constituye la unidad mínima procesable de una matriz tridimensional y es, por tanto, la extensión tridimensional del píxel bidimensional.
- **Kinect.** Sensor para videojuegos creado por Microsoft que ha sido utilizado en tareas de investigación para la reconstrucción de nubes de puntos de interiores. Microsoft provee un SDK (Software Development Kit) para poder programar este dispositivo y utilizarlo en otras aplicaciones.
- ***Dataset*.** Habitualmente se utiliza este término para referirse a un conjunto de datos preparado para realizar una tarea concreta, normalmente, algoritmos de aprendizaje automático. No existe una especificación formal para su estructura, siguiendo un estilo libre en función de la naturaleza del problema a resolver o el propósito del mismo.
- ***Grid*.** Se hace referencia a la expresión *grid* para referirse a un objeto tridimensional compuesto de un conjunto de celdas. Se trata de la extensión tridimensional de una matriz al uso: un *array* tridimensional.
- **Reflectividad.** Mide la capacidad de una superficie para reflejar la luz que incide sobre ella. Esta magnitud física suele medirse, en el contexto del

presente trabajo, utilizando un sensor LiDAR que hace referencia a ella mediante el concepto **intensidad**.

- **Conjunto de datos balanceado.** Al trabajar con problemas de clasificación en aprendizaje automático, es necesario partir de un conjunto de datos para realizar el entrenamiento y posterior validación. Para ello, el conjunto de datos de partida ha de estar balanceado, esto es, el número de ejemplos de cada clase del problema ha de ser equitativo con respecto al resto de clases. De lo contrario, el clasificador tenderá a clasificar posteriormente los ejemplos como la clase mayoritaria siempre, y no funcionará correctamente.
- **Imagen multiespectral.** Una imagen multiespectral es aquella que, además de capturar la información contenida en la radiación visible del espectro electromagnético, aporta otras bandas fuera del umbral visible.

7 BIBLIOGRAFÍA

Amari, S. (1993). Backpropagation and stochastic gradient descent method. *Neurocomputing*, *5*, 185–196.

Bengio, Y., LeCun, Y., & Henderson, D. (1993). Globally Trained Handwritten Word Recognizer Using Spatial Representation, Convolutional Neural Networks, and Hidden Markov Models. *Advances in Neural Information Processing Systems 6, [7th {NIPS} Conference, Denver, Colorado, USA, 1993]*, 937–944. Retrieved from http://papers.nips.cc/paper/819-globally-trained-handwritten-word-recognizer-using-spatial-representation-convolutional-neural-networks-and-hidden-markov-models

Bin Sulaiman, R. (2018). Artificial Intelligence Based Autonomous Car. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.3167638

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., & Vandergheynst, P. (2017). Geometric Deep Learning: Going beyond Euclidean data. *{IEEE} Signal Process. Mag.*, *34*(4), 18–42. https://doi.org/10.1109/MSP.2017.2693418

Cun, Y. Le. (1988). *A Theoretical Framework for Back-Propagation*.

Curry, E., Dustdar, S., Sheng, Q. Z., & Sheth, A. (2016). Smart cities – enabling services and applications. *Journal of Internet Services and Applications*, *7*(1), 6. https://doi.org/10.1186/s13174-016-0048-6

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., … Bengio, Y. (2014). Generative Adversarial Nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 27* (pp. 2672–2680). Retrieved from http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

Huang, J., & You, S. (2016). *Point Cloud Labeling using 3D Convolutional Neural Network*.

Jain, L. C., & Medsker, L. R. (1999). *Recurrent Neural Networks: Design and Applications* (1st ed.). Boca Raton, FL, USA: CRC Press, Inc.

Kingma, D., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 25* (pp. 1097–1105). Retrieved from http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Maturana, D., & Scherer, S. (2015). VoxNet: A 3D Convolutional Neural Network for Real-Time Object Recognition. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 922 – 928.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in

nervous activity. *The Bulletin of Mathematical Biophysics*, *5*(4), 115–133. https://doi.org/10.1007/BF02478259

Nguyen, A. T. Le, & Le, H. B. (2013). 3D point cloud segmentation: A survey. *2013 6th IEEE Conference on Robotics, Automation and Mechatronics (RAM)*, 225–230.

Qi, Charles R, Su, H., Mo, K., & Guibas, L. J. (2016). PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. *ArXiv Preprint ArXiv:1612.00593*.

Qi, Charles Ruizhongtai, Yi, L., Su, H., & Guibas, L. J. (2017). PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. In *Advances in Neural Information Processing Systems 30* (pp. 5099–5108). Curran Associates, Inc.

Ruder, S. (2016). *An overview of gradient descent optimization algorithms.* Retrieved from http://arxiv.org/abs/1609.04747

Russell, S. J., & Norvig, P. (2004). *Inteligencia artificial: un enfoque moderno*. Retrieved from https://books.google.es/books?id=yZCVPwAACAAJ

S. Pressman, R. (2006). *Ingeniería del Software Un Enfoque Práctico*.

Schmidhuber, J. (2015). Deep Learning in Neural Networks: An Overview. *Neural Networks*, *61*, 85–117.

Shelhamer, E., Long, J., & Darrell, T. (2017). Fully Convolutional Networks for Semantic Segmentation. *{IEEE} Trans. Pattern Anal. Mach. Intell.*, *39*(4), 640–651. https://doi.org/10.1109/TPAMI.2016.2572683

Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *International Conference on Learning Representations*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., … Rabinovich, A. (2015). Going Deeper with Convolutions. *CVPR*.

Wu, B., Wan, A., Yue, X., & Keutzer, K. (2018). SqueezeSeg: Convolutional Neural Nets with Recurrent {CRF} for Real-Time Road-Object Segmentation from 3D LiDAR Point Cloud. *2018 {IEEE} International Conference on Robotics and Automation, {ICRA} 2018, Brisbane, Australia, May 21-25, 2018*, 1887–1893. https://doi.org/10.1109/ICRA.2018.8462926